Tlgm: @it_boooks

O'REILLY®

«Джейсон, Скотт и Мэтт внесли основной вклад в обучение сетевых инженеров как в области сетевой автоматизации, так и в организации сетевой среды в ОС Linux».
– Кёрк Байерс, создатель библиотеки Netmiko Python Library

Сначала системные администраторы, а впоследствии и сетевые инженеры поняли, что они больше не могут выполнять свою работу вручную. Постоянное появление новых протоколов, технологий, моделей доставки и ужесточение требований к интеллектуальности и гибкости бизнес-процессов сделали сетевую автоматизацию чрезвычайно важной. Это практическое руководство наглядно демонстрирует сетевым инженерам, как использовать широкий спектр технологий и инструментальных средств, в том числе Linux, Python, JSON и XML, для автоматизации систем с помощью написания программного кода. Книга поможет вам упростить выполнение задач, связанных с конфигурированием, управлением и эксплуатацией сетевого оборудования, топологий, сервисов и поддержкой сетевых соединений. Внимательно изучая ее, вы получите основные практические навыки и освоите инструментальные средства, необходимые для сложного перехода к автоматизации сети.

Основные темы книги:

- основы программирования на Python: типы данных, условные выражения, циклы, функции, классы и модули;
- форматы и модели данных: JSON, XML, YAML и YANG для сети;
- роль прикладных программных интерфейсов (API) в сетевой автоматизации;
- способы практического применения средств автоматизации с открытым исходным кодом Ansible, Salt и StackStorm для автоматизации сетевых устройств.

Джейсон Эделман (Jason Edelman) *— основатель Network to Code, помогает клиентам адаптировать и развертывать инструментальные средства и технологии сетевой автоматизации. Он является активным пропагандистом сетевой автоматизации и совмещения практик DevOps и сетевых операций с 2013 года.*

Скотт С. Лоу (Scott S. Lowe) *— специалист по проектированию архитектуры VMware, Inc. Основной сферой его интересов являются облачные вычисления и виртуализация сетей. Скотт написал несколько технических книг по темам, связанным с использованием VMware vSphere и OpenStack.*

Мэтт Осуолт (Matt Oswalt) *— разработчик сетевого ПО, уделяющий главное внимание совмещению разработки ПО и сетевой инфраструктуры. Свои работы по этой теме и другие материалы он публикует на keepingitclassless.net.*

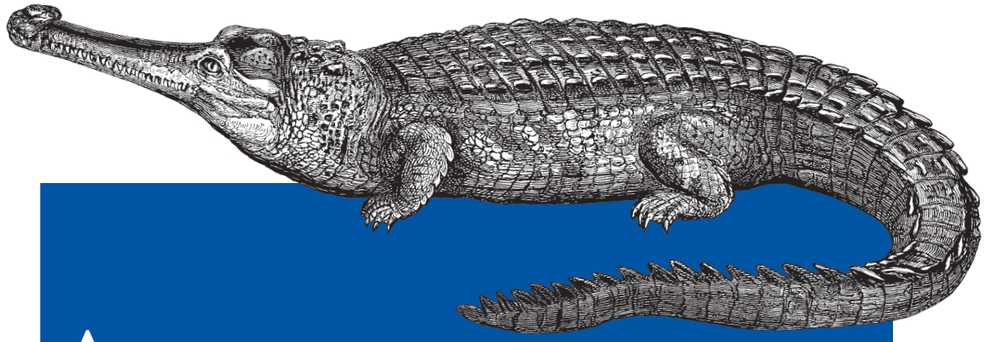
Интернет-магазин: www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа: "Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

ISBN 978-5-97060-699-5

9 785970 606995 >

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Автоматизация программируемых сетей



Автоматизация программируемых сетей

Джейсон Эделман
Скотт С. Лоу
Мэтт Осуолт

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Tlgm: @it_boooks

Джейсон Эделман, Скотт С. Лоу, Мэтт Осуолт

Автоматизация программируемых сетей

Tlgm: @it_boooks

Jason Edelman, Scott S. Lowe, and Matt Oswalt

Network Programmability and Automation

Skills for the Next-Generation Network Engineer

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Tlgm: @it_boooks

Джейсон Эделман, Скотт С. Лоу, Мэтт Осуолт

Автоматизация программируемых сетей

Профессиональная квалификация сетевого инженера нового поколения

ДМК Пресс
издательство
Москва, 2019

Tlgm: @it_boooks

УДК 004.5
ББК 32.812
Э30

Эделман Дж., Лоу С. С., Осуолт М.
Э30 Автоматизация программируемых сетей / пер. с анг. А. В. Снастина. – М.: ДМК Пресс, 2019. – 616 с.: ил.

ISBN 978-5-97060-699-5

Постоянное появление новых протоколов, технологий, моделей доставки и ужесточение требований к интеллектуальности и гибкости бизнес-процессов сделали сетевую автоматизацию чрезвычайно важной. Это практическое руководство наглядно демонстрирует сетевым инженерам, как использовать широкий спектр технологий и инструментальных средств, в том числе Linux, Python, JSON и XML, для автоматизации систем с помощью написания программного кода.

Книга поможет вам упростить выполнение задач, связанных с конфигурированием, управлением и эксплуатацией сетевого оборудования, топологий, сервисов и поддержкой сетевых соединений. Внимательно изучая ее, вы получите основные практические навыки и освоите инструментальные средства, необходимые для сложного перехода к автоматизации сети.

УДК 004.5
ББК 32.812

Authorized Russian translation of the English edition of Internet of Network Programmability and Automation ISBN 9781491931257 © 2018 Jason Edelman, Scott S. Lowe, Matt Oswalt.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-491-93125-7 (анг.)
ISBN 978-5-97060-699-5 (рус.)

© 2018 Jason Edelman, Scott S. Lowe, Matt Oswalt
© Оформление, издание, перевод, ДМК Пресс, 2019

Tlgm: @it_boooks

Эту книгу я посвящаю всем сетевым инженерам, начинающим свою деятельность по автоматизации сети. Я искренне надеюсь, что книга даст вам знания, необходимые для дальнейшего роста и развития. Также я хотел бы поблагодарить Скотта, Мэтта и всю рабочую группу издательства O'Reilly – понимаю, что процесс написания книги оказался более долгим, чем все мы планировали, но в итоге мы сделали это. Спасибо всем, кто сделал замысел реальностью.

– Джейсон Эделман (Jason Edelman)

Я хотел бы посвятить эту книгу Господу, давшему мне мудрость и знания, необходимые для ее написания (Исход 31:3, NIV). Также посвящаю свой труд моей жене Кристэл, без поддержки которой создание этой книги и многие другие вещи были бы невозможными.

– Скотт С. Лоу (Scott S. Lowe)

Я посвящаю эту книгу всем, кто жаждет новых знаний и всегда готов учиться с увлечением – каждое слово в книге написано для вас. Благодарю свою жену Джейми, которая поддерживала мое рабочее настроение и сохраняла нашу бодрость и жизнерадостность, даже когда жизнь становилась слегка сумасшедшей.

– Мэтт Осуолт (Matt Oswalt)

Tlgm: @it_boooks

Содержание

Предисловие 12

Глава 1. Тенденции в современной промышленной эксплуатации сетей 20

Возникновение технологии программно определяемой сети 20
- OpenFlow 21
- Что такое программно определяемая сеть 25

Резюме 39

Глава 2. Автоматизация сети 40

Для чего нужна автоматизация сети 40
- Упрощение архитектуры 41
- Детерминированные результаты 42
- Гибкость бизнеса 42

Типы автоматизации сети 43
- Подготовка и настройка устройств 43
- Сбор данных 46
- Переходы между платформами 47
- Управление конфигурацией 49
- Совместимость 49
- Составление отчетов 50
- Устранение проблем 51

Развитие уровня управления от протокола SNMP до API устройств 53
- Прикладные программные интерфейсы (API) 53
- Влияние концепции открытых сетей 57

Автоматизация сети в эпоху SDN 58

Резюме 59

Глава 3. Операционная система Linux 60

Изучение ОС Linux с точки зрения автоматизации сети 60

Краткая история создания ОС Linux 61

Дистрибутивы Linux 62
- Red Hat Enterprise Linux, Fedora и CentOS 62
- Debian, Ubuntu и другие производные дистрибутивы 64
- Другие дистрибутивы Linux 66

Работа в ОС Linux 66
- Перемещение по файловой системе 67
- Работа с файлами и каталогами 72
- Выполнение программ 79

Tlgm: @it_boooks

Работа с демонами ... 82
Работа с сетями в ОС Linux ... 87
Работа с интерфейсами ... 87
Маршрутизация для конечного хоста ... 98
Конфигурация маршрутизатора ... 103
Коммутация ... 105
Резюме ... 111

Глава 4. Изучение языка программирования Python для применения в сетевой среде ... 112

Должны ли сетевые инженеры уметь писать программный код? ... 113
Использование интерактивного интерпретатора Python ... 115
Типы данных языка Python ... 117
Использование строк ... 118
Использование числовых значений ... 128
Использование логических значений ... 130
Использование списков ... 133
Использование словарей ... 138
Множества и кортежи языка Python ... 143
Использование условных логических выражений ... 145
Концепция объекта, содержащего другие объекты ... 148
Использование циклов ... 149
Использование цикла while ... 149
Использование цикла for ... 150
Использование функций ... 154
Работа с файлами ... 158
Чтение данных из файла ... 158
Запись данных в файл ... 161
Создание программ на языке Python ... 163
Создание простого скрипта на языке Python ... 163
Что такое shebang ... 164
Перемещение кода из интерпретатора Python в независимый скрипт ... 166
Работа с модулями языка Python ... 167
Передача аргументов в скрипт ... 169
Использование pip для установки пакетов языка Python ... 171
Советы, приемы и дополнительная информация по использованию языка Python ... 173
Резюме ... 179

Глава 5. Форматы и модели данных ... 180

Введение в форматы данных ... 180
Типы данных ... 182
YAML ... 184
Краткий обзор основ YAML ... 184
Работа с YAML в коде Python ... 187
Модели данных в YAML ... 188
XML ... 190

Tlgm: @it_boooks

Основы XML ... 190
Использование определения схемы XML Schema Definition (XSD) для моделей данных ... 191
Преобразование XML с помощью XSLT ... 193
Поиск в данных XML с использованием XQuery ... 197
JSON ... 197
Основы формата JSON ... 198
Обработка формата JSON в коде Python ... 200
Использование механизма JSON Schema для моделей данных ... 201
Создание моделей данных с использованием YANG ... 202
Общий обзор языка YANG ... 202
Практическое применение языка YANG ... 203
Резюме ... 207

Глава 6. Шаблоны сетевой конфигурации ... 208

Современные языки шаблонов ... 209
Использование шаблонов для веб-разработки ... 210
Универсальность шаблонов ... 211
Важность использования шаблонов в процессе автоматизации сети ... 212
Язык Jinja для создания шаблонов сетевой конфигурации ... 213
Почему именно Jinja ... 213
Динамическая вставка данных в простой шаблон Jinja ... 214
Обработка файла шаблона Jinja средствами языка Python ... 215
Условные выражения и циклы ... 217
Фильтры Jinja ... 224
Наследование шаблонов в языке Jinja ... 227
Создание переменных в Jinja ... 228
Резюме ... 229

Глава 7. Использование сетевых прикладных программных интерфейсов (API) ... 230

Основы сетевых API ... 231
Введение в API-интерфейсы на основе протокола HTTP ... 231
Основы NETCONF ... 236
Практическое использование сетевых API ... 244
Практическое использование API на основе протокола HTTP ... 245
Практическое использование NETCONF ... 252
Автоматизация с использованием сетевых API ... 261
Использование библиотеки requests ... 262
Использование Python-библиотеки ncclient ... 292
Использование библиотеки netmiko ... 317
Резюме ... 322

Глава 8. Управление исходным кодом с помощью Git ... 325

Варианты использования средств управления исходным кодом ... 326
Преимущества системы управления исходным кодом ... 326

Tlgm: @it_boooks

Отслеживание изменений ... 327
Учетные записи ... 327
Процесс и рабочий поток ... 327
Преимущества системы управления исходным кодом в сетевой среде ... 328
Знакомство с Git ... 328
Краткая история создания и развития Git ... 329
Терминология Git ... 330
Обзор архитектуры Git ... 331
Работа с системой Git ... 332
Установка системы Git ... 332
Создание репозитория ... 333
Добавление файлов в репозиторий ... 333
Выполнение коммита изменений в репозиторий ... 335
Внесение изменений и выполнение коммитов в отслеживаемые файлы ... 339
Отмена фиксации файлов в индексе ... 342
Исключение файлов из репозитория ... 345
Получение более подробной информации о репозитории ... 349
Определение различий между версиями файлов ... 354
Создание ветвей версий в системе Git ... 358
Создание ветви ... 363
Выбор активной ветви ... 364
Объединение и удаление ветвей ... 366
Совместная работа группы сотрудников в системе Git ... 371
Совместная работа в нескольких системах, использующих Git ... 372
Совместная работа с использованием онлайновых сервисов на основе Git ... 389
Резюме ... 395

Глава 9. Инструментальные средства автоматизации ... 396

Краткий обзор инструментальных средств автоматизации ... 396
Использование Ansible ... 399
Основы работы Ansible ... 400
Создание inventory-файла ... 401
Выполнение сценария Ansible ... 408
Использование файлов переменных ... 412
Создание комплектов сценариев Ansible для автоматизации сети ... 414
Использование сторонних модулей Ansible от независимых авторов ... 433
Резюме по системе Ansible ... 436
Автоматизация сети с использованием Salt ... 437
Основы архитектуры Salt ... 437
Общая информация о Salt ... 440
Управление сетевыми конфигурациями с помощью Salt ... 458
Удаленное выполнение функций Salt ... 467
Управляемая событиями инфраструктура Salt ... 469
Дополнительная информация о Salt ... 475
Краткий итоговый обзор системы Salt ... 478
Автоматизация сети, управляемая событиями, с использованием StackStorm ... 479
Основные концепции системы StackStorm ... 480

Tlgm: @it_boooks

Архитектура StackStorm 482
Операции и рабочие потоки 484
Сенсоры и триггеры 493
Правила 496
Краткий итоговый обзор системы StackStorm 499
Резюме 499

Глава 10. Непрерывная интеграция 500
Важные предпосылки 502
Чем проще, тем лучше 502
Люди, процесс и технология 503
Изучение программного кода 503
Введение в непрерывную интеграцию 504
Основы непрерывной интеграции 504
Непрерывная доставка 506
Разработка через тестирование 508
Применимость методики непрерывной интеграции к сетевой среде 511
Конвейер непрерывной интеграции для сетевой среды 512
Рецензирование коллегами 513
Автоматизация сборки 519
Среда тестирования/разработки/перемещения данных 524
Инструментальные средства развертывания 528
Инструментальные средства тестирования и автоматизация сети по методике разработки через тестирование 531
Резюме 533

Глава 11. Формирование культуры автоматизации сети 535
Организационная стратегия и гибкость 536
Преобразование организации старого образца 536
Важность поддержки со стороны руководства 538
Купить или создать самостоятельно 540
Восприятие ситуаций критических сбоев 541
Практические навыки и обучение 543
Изучайте неизвестное 544
Сосредоточьтесь на основных принципах 545
Нужны ли сертификации? 546
Может ли автоматизация лишить людей работы 547
Резюме 548

Приложение А. Профессиональное управление сетевой средой в ОС Linux 550
Использование интерфейсов macvlan 550
Варианты практического использования интерфейсов macvlan 551
Создание, конфигурирование и удаление интерфейсов macvlan 551
Виртуальные машины в сетевой среде 553
Использование шлюза 554

Tlgm: @it_boooks

Использование интерфейсов macvtap ... 557
Работа с сетевыми пространствами имен ... 558
Практические примеры использования сетевых пространств имен ... 559
Создание и удаление сетевых пространств имен ... 560
Размещение интерфейсов в сетевом пространстве имен ... 560
Выполнение команд в определенном сетевом пространстве имен ... 562
Соединение сетевых пространств имен с помощью пар veth ... 564
Использование контейнеров Linux в сетевой среде ... 566
Конфигурирование сетевой среды в LXC ... 567
Конфигурирование сетевой среды в Docker ... 568
Использование Open vSwitch ... 570
Установка OVS ... 570
Конфигурирование OVS ... 572
Соединение нескольких типов рабочих нагрузок в OVS ... 575

Приложение Б. Использование NAPALM ... 583

Управление конфигурацией с использованием NAPALM ... 583
Выполнение операции замены конфигурации ... 584
Выполнение операции объединения конфигураций ... 588
Получение данных от устройств с помощью NAPALM ... 591
Возможности интеграции NAPALM с другим ПО ... 593
Использование NAPALM в Ansible ... 594
Использование NAPALM в Salt ... 595
Использование NAPALM в StackStorm ... 596

Предметный указатель ... 598

Tlgm: @it_boooks

Предисловие

Приветствуем всех читателей книги «Программно управляемые сети и их автоматизация».

Изменения в области промышленной эксплуатации сетей постоянны и значительны. Ориентация организаций и специалистов-профессионалов по сетям на восприятие идей и концепций сетевой программируемости и автоматизации сейчас приобретает гораздо большее значение, чем раньше, и стимулируется непрерывным процессом появления новых протоколов, новых технологий, новых моделей доставки, а также потребностями бизнес-процессов в плане большей интеллектуальности и гибкости для обеспечения конкурентоспособности. Но что означает сетевая программируемость и автоматизация? Начнем эту книгу с попытки найти возможности ответа на данный вопрос.

О чем эта книга

В соответствии с названием главной темой книги является сетевая программируемость (программно управляемые сети) и автоматизация. По существу, сетевая программируемость и автоматизация практически является синонимом упрощения задач по конфигурированию, управлению и эксплуатации сетевого оборудования, сетевых топологий и поддержки сетевых соединений. При этом используется множество разнообразных компонентов, в том числе операционные системы, которые в настоящее время гораздо шире применяются в сетевых средах, чем в прошлом, новые методики, например методика непрерывной интеграции (continuous integration), и привлечение инструментальных средств, прежде относящихся только к арсеналу системного администратора (например, средства управления версиями исходного кода и системы управления конфигурацией). Мы полагаем, что все перечисленные компоненты играют важную роль в базовом определении сущности сетевой программируемости и автоматизации, поэтому рассматриваем все эти темы. Цель данной книги – позволить читателям получить основы знаний о сетевой программируемости и автоматизации.

Как организована эта книга

Эту книгу не обязательно штудировать последовательно, страницу за страницей, мы разделили ее на главы таким образом, чтобы читатель с легкостью находил наиболее интересные ему темы. Возможно, правильнее всего будет начать с последовательного чтения первых трех глав, так как в них представлена общая информация, то есть фактически закладывается основа понимания

Tlgm: @it_boooks

всей остальной части книги. Далее вы можете свободно переходить к темам, которые вам наиболее интересны или немедленно требуются для практической работы. Мы пытались сохранить относительную независимость глав друг от друга, но, как и в процессе применения любой технологии, это оказалось не всегда выполнимо. Везде, где это возможно, мы приводим перекрестные ссылки, чтобы помочь читателям быстро находить необходимую информацию.

Ниже приводится краткий обзор организации книги по главам:

- глава 1 «Тенденции в современной промышленной эксплуатации сетей» представляет обзор основных событий и тенденций, способствовавших появлению программно определяемых сетей (software defined networks, SDN). Как вы увидите в главе 1, программно определяемая (или конфигурируемая) сеть стала главной первопричиной постоянно возрастающего сосредоточения внимания на сетевой программируемости и автоматизации;
- глава 2 «Автоматизация сети» продолжает обсуждение программно определяемых сетей, начатое в главе 1, но здесь главное внимание уделяется автоматизации сети: истории развития, влиянию сетевой автоматизации на операционные (рабочие) модели (и влиянию операционных моделей на автоматизацию);
- глава 3 «ОС Linux» представляет общий обзор операционной системы Linux. Эта глава не является полным описанием ОС Linux, ее основная задача – быстро ознакомить специалистов по сетям с рабочей средой Linux, с основным набором команд и с сетевыми концепциями, реализованными в Linux;
- глава 4 «Изучение языка программирования Python в сетевом контексте» знакомит специалистов по сетям с языком программирования Python (http://python.org/). Язык Python часто применяется для программируемых сетей и их автоматизации, поэтому в главе рассматриваются многие основные аспекты программирования на Python: типы данных, условные выражения, циклы, работа с файлами, функции, классы и модули;
- глава 5 «Форматы и модели данных» представляет наиболее распространенные форматы данных, которые часто встречаются в проектах автоматизации сетей. Рассматриваются языки обработки данных JavaScript Object Notation (JSON), eXtensible Markup Language (XML), YAML Ain't Markup Language (YAML). Затем описываются концепции моделирования данных и приводится краткое введение в язык YANG, широко распространенный язык моделирования данных для сетевой среды;



Что означает термин «формат данных»

Если для вас это незнакомый термин, не смущайтесь. Формат данных (data format) – это всего лишь способ кодирования или инкапсуляции данных при передаче между двумя пунктами (например, при возврате данных в ответ на вызов функции прикладного программного интерфейса (API)). Форматы данных подробно рассматриваются в главе 5.

Tlgm: @it_boooks

- глава 6 «Шаблоны конфигурации сети» – рассматривается использование языков описания шаблонов для создания конфигураций сетевых устройств. Основное внимание сосредоточено на языке шаблонов Jinja, так как он весьма удачно взаимодействует с языком программирования (ЯП) Python. Кроме того, рассматриваются еще два языка описания шаблонов: Mako и ERB. Mako интегрируется с ЯП Python, а ERB используется в основном совместно с ЯП Ruby;
- глава 7 «Работа с сетевыми прикладными программными интерфейсами (API)» – рассматривается роль прикладных программных интерфейсов (application programming interfaces, API) в обеспечении сетевой программируемости и автоматизации. Описываются основные термины и технологии, применяемые в API, а также применение некоторых наиболее распространенных API от конкретных производителей – API для устройств и API для контроллеров – с примерами, наглядно демонстрирующими возможности их использования для обеспечения сетевой программируемости и автоматизации;
- глава 8 «Управление исходными кодами с помощью Git» представляет систему Git (https://git-scm.com) – общеизвестное и широко используемое инструментальное средство управления исходными кодами. Подчеркивается важность управления исходными кодами, рассматриваются возможности использования такой системы с точки зрения сетевой программируемости и автоматизации. Также рассматривается работа с известными онлайновыми сервисами, например GitHub (https://github.com);
- глава 9 «Инструментальные средства автоматизации» подробно описывает использование инструментальных средств автоматизации с открытым исходным кодом, таких как Ansible (http://www.ansible.com/home), Salt (http://saltstack.com) и StackStorm (https://stackstorm.com), и способы применения этих инструментов специально для обеспечения сетевой программируемости и автоматизации;
- глава 10 «Непрерывная интеграция» описывает концепции непрерывной интеграции (continuous integration, CI) и соответствующие основные инструментальные средства и технологии. Рассматривается применение методики разработки через тестирование (test-driven development, TDD), инструментов исследования и контроля, а также комплексных сред разработки, таких как Jenkins и Gerrit, приводится пример формирования потока сетевой автоматизации, включающий все элементы методики непрерывной интеграции;
- глава 11 «Формирование культуры автоматизации сети» объясняет, почему правильно сформированная культура чрезвычайно важна и почему культура является основополагающим элементом для автоматизации сети. Рассматривается процесс формирования такой культуры;
- приложение А «Дополнительная информация о поддержке сетевой среды в ОС Linux» продолжает обсуждение, начатое в главе 3, но с более подробными описаниями работы с сетевыми интерфейсами macvlan, орга-

Tlgm: @it_boooks

низации сетей с помощью виртуальных машин (VM), работы с сетевыми пространствами имен и с сетями, использующими контейнеры Linux (включая и контейнеры Docker (https://www.docker.com)). Также рассматривается использование Open vSwitch (OVS) (http://openvswitch.org);

- приложение Б «Использование NAPALM» представляет введение в использование библиотеки NAPALM (Network Automation and Programmability Abstraction Layer with Multi-vendor support), написанной на ЯП Python. Рассматривается применение NAPALM для управления конфигурацией, не зависящей от конкретных производителей, а также для извлечения данных из сетевых устройств. Кратко описываются возможности интеграции NAPALM с инструментальными средствами Ansible, Salt и StackStorm, которые подробно рассматривались в главе 9.

Для кого предназначена эта книга

Как уже было сказано ранее, главная цель этой книги – предоставить читателям основополагающие знания и набор основных практических навыков в области сетевой программируемости и автоматизации. Мы надеемся, что представители различных профессий в области информационных технологий извлекут пользу из чтения нашей книги.

Сетевые инженеры

Вполне естественно, что бóльшую часть читателей книги, посвященной сетевой программируемости и автоматизации, составляют «обычные» сетевые инженеры, то есть специалисты, достаточно хорошо разбирающиеся в сетевых протоколах, в конфигурации сетевых устройств и в вопросах функционирования и управления сетями. Мы надеемся, что книга поможет сетевым инженерам сделать свою повседневную работу более эффективной и производительной с помощью автоматизации и практического использования программируемости сетей.

Предварительные требования

Сетевым инженерам, заинтересованным в более глубоком изучении сетевой программируемости и автоматизации, не требуются какие-либо предварительные знания в области разработки ПО, программирования, автоматизации или инструментальных средств, связанных с DevOps. Единственное предварительное требование – свободное мышление и восприятие без предвзятости и желание изучать новые технологии и их влияние на деятельность специалистов по сетям и на всю сетевую индустрию в целом.

Системные администраторы

Системные администраторы, ответственные в основном за управление системами, подключенными к сетям, возможно, уже обладают некоторым предварительным уровнем знаний и практическим опытом использования инструмен-

Tlgm: @it_boooks

тальных средств, рассматриваемых в книге (в особенности ОС Linux, систем управления исходным кодом и систем управления конфигурацией). Таким образом, книга может помочь им расширить знания и понимание функциональности таких инструментов, представляя их в различных контекстах (например, использование Ansible для конфигурирования сетевого коммутатора в сравнении с использованием Ansible для конфигурирования сервера, работающего под управлением дистрибутива Linux).

Предварительные требования

В этой книге не рассматриваются и не объясняются базовые сетевые протоколы и концепции. Но мы предполагаем, что многие системные администраторы, постоянно занимающиеся управлением подключенных к сетям систем, обладают достаточным уровнем знаний об основных сетевых протоколах, поэтому хорошо подготовлены к чтению. Если вы не вполне уверены в качестве своих знаний о работе в сетях, рекомендуем предварительно изучить одну из книг, подробно описывающих основные сетевые концепции и принципы функционирования. Например, неплохим вариантом является книга «Packet Guide to Core Network Protocols» издательства O'Reilly.

Разработчики программного обеспечения

Разработчики программного обеспечения также могут получить много полезной информации при чтении этой книги. Многие разработчики уже обладают опытом работы с языками программирования и средствами разработки, обсуждаемыми в книге (например, Python и Git). Как и для системных администраторов, для разработчиков может оказаться полезным рассмотрение использования средств разработки и языков программирования в применении к сетевым средам (например, описание возможностей Python для извлечения и сохранения специализированных сетевых данных).

Предварительные требования

Предполагается, что читатели обладают базовыми знаниями об основных сетевых протоколах и концепциях, поскольку все предлагаемые в книге примеры выполняются исключительно в сетевой среде. Если для разработчиков ПО сети являются новой темой, то, как и системным администраторам, мы предлагаем предварительно изучить одну из книг по основным сетевым концепциям.

Инструменты, используемые в книге

В области сетевой программируемости и автоматизации, как и в любой технической отрасли, существует множество версий и вариаций технологий и инструментальных средств. Поэтому мы придерживаемся общепризнанных стандартов при рассмотрении инструментальных средств в данной книге и выбираем самые лучшие, по нашему мнению, из этих инструментов, чтобы читатели сами оценили их полезность для своей практической деятельности. Например, при

Tlgm: @it_boooks

наличии множества разнообразных дистрибутивов Linux основное внимание мы все же уделяем дистрибутивам Debian, Ubuntu (производному от Debian) и CentOS (производному от Red Hat Enterprise Linux (RHEL)). Чтобы облегчить восприятие информации для читателей, мы рассматриваем конкретную версию инструментального средства в каждой соответствующей главе.

Онлайн-ресурсы

Мы понимаем, что в одной книге невозможно представить весь материал, полностью освещающий такую обширную тему, как сетевая программируемость и автоматизация. Поэтому на протяжении всей книги мы многократно ссылаемся на дополнительные онлайн-ресурсы, которые могут оказаться весьма полезными при более глубоком изучении концепций, принципов и практических навыков, рассматриваемых в этой книге.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения:

Курсив – используется для смыслового выделения важных положений, новых терминов, URL-адресов и адресов электронной почты в интернете, имен команд и утилит, а также имен и расширений файлов и каталогов.

`Моноширинный шрифт` – используется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт` – используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

`Моноширинный курсив` – используется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.



Такая пиктограмма обозначает указание или примечание общего характера.



Эта пиктограмма обозначает предупреждение или особое внимание к потенциально опасным объектам.



Такая пиктограмма обозначает совет или рекомендацию.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Tlgm: @it_boooks

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от огорчения и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Благодарности

Создание этой книги было бы невозможным без помощи и поддержки большой группы людей.

Tlgm: @it_boooks

Во-первых, мы хотели бы выразить свою благодарность чрезвычайно активному сообществу людей, объединенных идеей сетевой автоматизации. Назвать всех по именам невозможно, потому что их слишком много, но эти люди создали проекты с открытым исходным кодом, такие как NAPALM и Netmiko, всячески способствовали вовлечению новичков в процесс обучения сетевой автоматизации и постоянно делились своими знаниями и опытом с другими. Спасибо всем участникам этого сообщества за их усилия и за участие в создании этой книги.

Сотрудничающие с нами авторы помогли сделать книгу более совершенной и всеобъемлющей, и мы не смогли бы добиться такого результата без их содействия, поэтому мы выражаем огромную благодарность за помощь. Мирча Улинич (Mircea Ulinic) написал раздел SaltStack для главы об инструментальных средствах управления сетевой конфигурацией, а Джере Джулиан (Jere Julian) предоставил материал о Puppet, который мы, к сожалению, не смогли включить в текущую версию книги. Спасибо, Мирча и Джере.

Технические рецензенты весьма серьезно относились к проверке содержимого книги, соблюдая баланс между точностью и правильностью излагаемого материала и легкостью восприятия этого материала читателем. Мы выражаем свою благодарность Патрику Огенстаду (Patrick Ogenstad), Ахилу Белу (Akhil Behl), Эрику Чоу (Eric Chou) и Сринивасу Макаму (Sreenivas Makam). Спасибо за то, что помогли!

Список людей, заслуживающих благодарности, был бы неполным без упоминания в нем сотрудников O'Reilly Media: Вирджиния Уилсон (Virginia Wilson) и Куртни Аллен (Courtney Allen) – редакторы, Дуайт Рэмси (Dwight Ramsey) – литературный редактор, Рэчел Монэхен (Rachel Monaghan) – корректор, Джуди МакКонвил (Judy McConville) – составитель предметного указателя, Колин Коул (Colleen Cole) – технический редактор, Рэнди Комер (Randy Comer) – дизайнер обложки и Ребекка Демаре (Rebecca Demarest) – художник-иллюстратор. Важность их помощи на всем пути книги от идеи до воплощения невозможно переоценить, и мы благодарим этих людей за их увлеченность работой и вклад в создание книги.

Tlgm: @it_boooks

Глава 1

Тенденции в современной промышленной эксплуатации сетей

Вам мало знаком термин «программно определяемая сеть» (Software Defined Networking, SDN)? Или вы захвачены массовым увлечением технологией SDN в последние несколько лет? В какую бы категорию вы ни попали, не стоит беспокоиться. Эта книга проведет вас по всем основным темам и поможет начать изучение сетевой программируемости и автоматизации с момента появления SDN. В данной главе представлен обзор тенденций в современной промышленной эксплуатации сетей, при этом особое внимание уделено SDN, ее значимости и воздействию этой технологии на современный мир сетей. Начнем мы с исторического обзора внедрения SDN в основной пул технологий и окончательного формирования направлений сетевой программируемости и автоматизации.

Возникновение технологии программно определяемой сети

Если бы нужно было назвать только одного человека, который изменил всю сетевую индустрию, это был бы Мартин Касадо (Martin Casado), в настоящее время являющийся главным партнером (General Partner) и вкладчиком-инвестором (Venture Capitalist) в венчурном фонде Andreessen Horowitz. Ранее Касадо был действительным членом совета VMware, первым вице-президентом

Tlgm: @it_boooks

и генеральным директором подразделения Networking and Security Business в компании VMware. Он оказал огромное воздействие на всю сетевую индустрию не только непосредственным участием в разработках (включая OpenFlow и Nicira), но и прояснением общей ситуации в отношении обязанностей, возлагаемых на крупные сети, а также острой необходимости внесения изменений в функционирование, адаптируемость и управляемость сетей. Рассмотрим эти факты более подробно.

OpenFlow

Как бы то ни было, OpenFlow послужил в качестве первого основного протокола в процессе внедрения и продвижения программно определяемой (или конфигурируемой) сети (Software Defined Network, SDN). Мартин Касадо разрабатывал протокол OpenFlow в процессе получения кандидатской степени (PhD) в Стэнфордском университете (Stanford University) под руководством Ника МакКеона (Nick McKeown). OpenFlow представляет собой протокол, который всего лишь позволяет отделить уровень управления сетевыми устройствами от уровня представления данных (см. рис. 1.1). Проще говоря, уровень управления можно интерпретировать как разум, мозг (brains) сетевого устройства, а уровень представления данных – как аппаратуру (hardware) или интегральную схему специального назначения (application-specific integrated circuits, ASIC), которая действительно выполняет перенаправление (forwarding) пакетов.

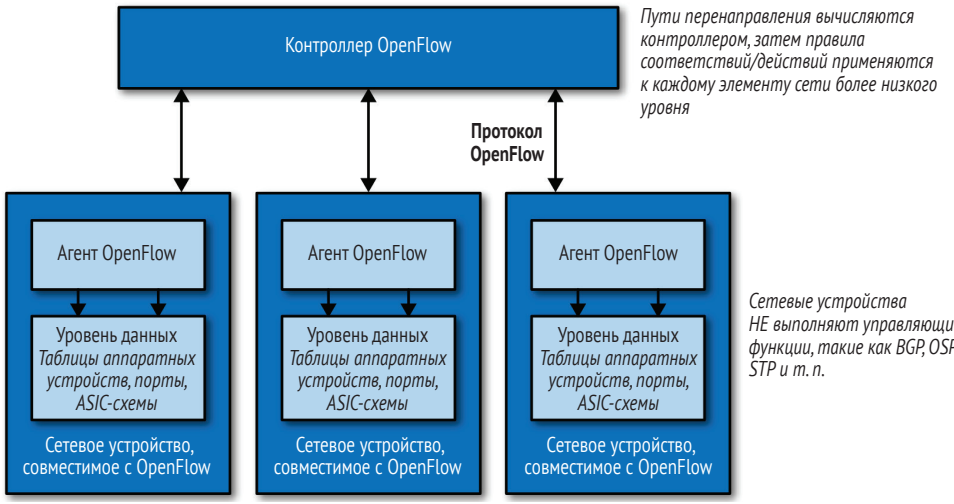

Контроллер OpenFlow
Пути перенаправления вычисляются контроллером, затем правила соответствий/действий применяются к каждому элементу сети более низкого уровня
Протокол OpenFlow
Агент OpenFlow
Уровень данных
Таблицы аппаратных устройств, порты, ASIC-схемы
Сетевое устройство, совместимое с OpenFlow
Агент OpenFlow
Уровень данных
Таблицы аппаратных устройств, порты, ASIC-схемы
Сетевое устройство, совместимое с OpenFlow
Агент OpenFlow
Уровень данных
Таблицы аппаратных устройств, порты, ASIC-схемы
Сетевое устройство, совместимое с OpenFlow
Сетевые устройства НЕ выполняют управляющие функции, такие как BGP, OSPF, STP и т. п.

Рис. 1.1 ❖ Разделение уровня управления и уровня данных с помощью протокола OpenFlow

Tlgm: @it_boooks

Работа OpenFlow в гибридном режиме

На рис. 1.1 показаны элементы сети, не имеющие уровня управления. Эта схема представляет «чистый» вариант развертывания на основе только протокола OpenFlow. Многие устройства также поддерживают работу OpenFlow в гибридном режиме, то есть протокол OpenFlow может быть развернут на заданном порту, в виртуальной локальной сети (VLAN) или даже в обычном канале перенаправления пакетов таким образом, что если в таблице OpenFlow не найдено соответствие, то используются существующие таблицы перенаправления (MAC-адреса, таблицы маршрутизации и т. п.). Такой режим работы в большей степени похож на маршрутизацию на основе правил (Policy Based Routing, PBR).

Все сказанное выше означает, что OpenFlow представляет собой протокол низкого уровня, используемый для прямого взаимодействия с таблицами аппаратных устройств (например, с информационной базой данных переадресации (Forwarding Information Base, FIB)), которые указывают сетевому устройству, как перенаправлять трафик (например, «трафик на целевой адрес 192.168.0.100 должен исходить из порта 48»).



OpenFlow – это протокол низкого уровня, который управляет таблицами потоков (данных), то есть напрямую воздействует на перенаправление (переадресацию) пакетов. OpenFlow не предназначен для взаимодействия с атрибутами уровня управления, такими как процедура аутентификации или параметры протокола SNMP.

Поскольку таблицы OpenFlow не ограничиваются поддержкой только целевых адресов, в отличие от обычных протоколов маршрутизации, они обеспечивают бóльшую детализацию (соответствие полей в пактах) при определении пути перенаправления. Но этот подход отличается от детализации, предлагаемой маршрутизацией на основе правил (PBR). Подобно OpenFlow в далеком будущем, PBR-маршрутизация позволяет сетевым администраторам перенаправлять трафик на основе «не совсем обычных» атрибутов, таких как адрес источника пакетов. Но при этом для поставщиков сетевого оборудования потребовалось некоторое время, чтобы обеспечить соответствующую производительность при PBR-маршрутизации трафика, поэтому окончательный результат оставался в весьма сильной зависимости от поставщиков. Появление протокола OpenFlow означало, что теперь можно достичь того же уровня детализации в принятии решений о переадресации, но при этом полностью устраняется зависимость от производителей и поставщиков оборудования. Появилась возможность расширить функциональные возможности сетевой инфраструктуры без ожидания следующей версии аппаратных устройств от производителей.

История программируемых сетей

OpenFlow не был самым первым протоколом или технологией, используемой для отделения функций управления и принятия решений от сетевых устройств. Его появлению предшествовала долгая история создания технологий и исследований, тем не менее

Tlgm: @it_boooks

именно OpenFlow стал технологией, начавшей «революцию программно определяемых сетей». Предшественниками OpenFlow являются такие технологии, как Forwarding and Control Element Separation (ForCES), Active Networks, Routing Control Platform (RCP) и Path Computation Element (PCE). Чтобы более подробно ознакомиться с историей развития программно определяемых сетей и соответствующих технологий, прочтите статью «The Road to SDN: An Intellectual History of Programmable Networks» (https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/sdnhistory.pdf), авторы: Джен Рексфорд (Jen Rexford), Ник Фимстер (Nick Feamster) и Элен Зегура (Ellen Zegura).

Почему именно OpenFlow?

Несмотря на важность понимания того, что представляет собой протокол OpenFlow, еще более важно понять основания и причины, по которым начались исследования и разработка первоначальных технических характеристик OpenFlow и последующее появление программно определяемых сетей.

Мартин Касадо работал на национальное правительство во время своего обучения в Стэнфордском университете. В процессе этой работы возникла необходимость организации ответных действий при атаках, угрожающих безопасности IT-систем (прежде всего IT-систем правительства США). Касадо быстро понял, что можно создать программу, которая управляет компьютерами и серверами именно так, как ему нужно. Реальные примеры использования этой программы никогда не публиковались, но это был тот тип управления конечными точками сети, который позволял предпринимать ответные действия, анализировать и в перспективе перепрограммировать хост или группу хостов в тех случаях, когда это необходимо.

В то время в реальных сетях было почти невозможно выполнить задуманное исключительно программным способом. Каждое сетевое устройство было «закрытым» (например, блокировалась установка любого стороннего программного обеспечения) и предлагало только интерфейс командной строки (command-line interface, CLI). Хотя интерфейс командной строки был и остается широко распространенным и даже предпочитаемым инструментом сетевых администраторов, Касадо ясно понял, что этот инструмент не способен обеспечить гибкость, требуемую для управления, эксплуатации и защиты сети.

В действительности за последние 20 лет способ управления сетями не изменился, за исключением добавления функциональных возможностей с помощью новых команд в интерфейс командной строки. Самым крупным изменением был переход с Telnet на SSH, послуживший основой для шутки, часто используемой SDN-компанией Big Switch Networks на своих демонстрационных слайдах, один из которых можно видеть на рис. 1.2.

Но если говорить серьезно, то технология управления сетями значительно отстала от других технологий, поэтому Касадо в конечном итоге занялся этой проблемой, чтобы изменить положение дел в течение нескольких следующих лет. Скудность средств управления часто лучше осознается при изучении других технологий. Другие технологии почти всегда предлагают более современ-

Tlgm: @it_boooks

ные способы управления большим количеством устройств и для управления конфигурацией, и для сбора и анализа данных – например, менеджеры гипервизоров, контроллеры беспроводных сетей, телефонные системы IP PBX, PowerShell, инструменты DevOps – список можно продолжать бесконечно. Некоторые из перечисленных средств являются коммерческим программным обеспечением, то есть собственностью производителя, но прочие инструменты можно свободно применять для управления, эксплуатации и обеспечения гибкости сетей независимо от конкретной платформы.

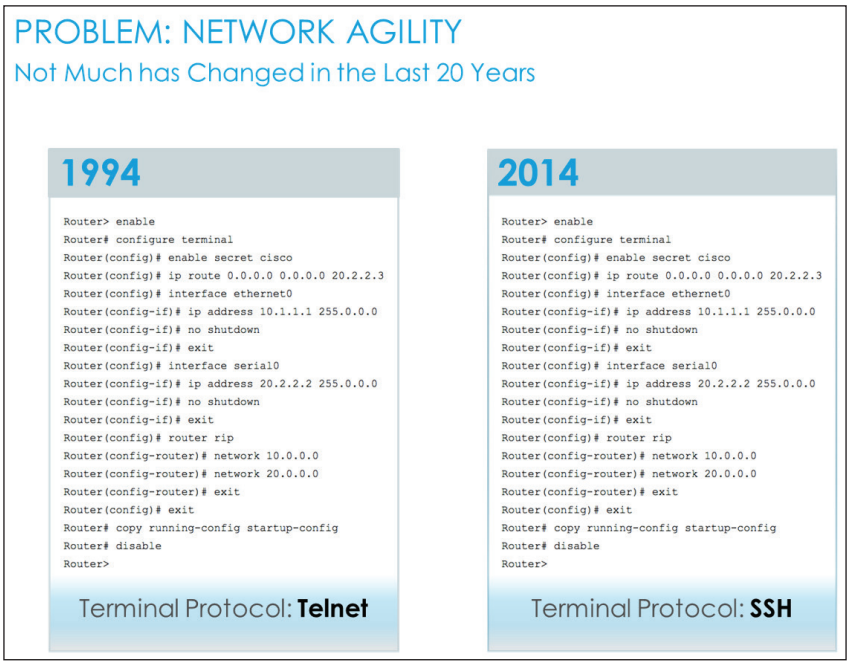


Рис. 1.2 ❖ Что изменилось? Telnet заменили на SSH (источник: Big Switch Networks)

Если вновь вернуться к ситуации, когда Касадо работал на правительство, то возникает вопрос: существовала ли возможность перенаправления трафика на уровне прикладного программного обеспечения? Существовали ли в то время прикладные программные интерфейсы (API) для сетевых устройств? Существовал ли единый пункт обмена информацией для всей сети? На все эти вопросы в большинстве случаев следовал отрицательный ответ. Каким образом появилась возможность программирования сети для динамического управления перенаправлением пакетов, стратегиями и конфигурациями с такой же простой реализацией, как написание программы и выполнение ее на оконечной хост-машине?

Tlgm: @it_boooks

Первоначальная спецификация OpenFlow стала результатом исследований Мартина Касадо, который искал пути решения перечисленных выше проблем. Когда ажиотаж вокруг OpenFlow прекратился, поскольку в итоге внимание сосредоточилось в большей степени на вариантах применения и практических решениях, нежели на протоколах низкого уровня, первая работа Касадо стала катализатором для переосмысления всего процесса создания, управления и эксплуатации сетей. Благодарим тебя, Мартин.

Кроме того, это значит, что если бы не было Мартина Касадо, то, возможно, эта книга никогда не была бы написана. Хотя кто знает…

Что такое программно определяемая сеть

В предыдущем разделе был кратко описан протокол OpenFlow, но что такое программно определяемая (или конфигурируемая) сеть (Software Defined Networking, SDN)? Это одно и то же или нет, а может быть, это совершенно разные вещи? Откровенно говоря, SDN – это практически то же самое, чем была облачная среда (Cloud) около десяти лет назад, до тех пор, пока мы не узнали о различных типах облака: инфраструктура как сервис (Infrastructure as a Service, IaaS), платформа как сервис (Platform as a Service, PaaS) и программное обеспечение как сервис (Software as a Service, SaaS).

Есть множество примеров и проектных решений, способствующих пониманию того, чем была облачная среда и чем она стала сейчас, но даже до того, как появились вышеперечисленные термины, могли возникать споры о том, что именно подразумевается при упоминании «облачной среды». Аналогичная ситуация и с термином «программно определяемая сеть». Существуют общедоступные определения, утверждающие, что сетевая среда как «прозрачный ящик» – это SDN или что наличие прикладного программного интерфейса (API) для сетевого устройства – это SDN. Но действительно ли такие характеристики определяют термин SDN? Разумеется, нет.

Вместо попыток определения SDN мы рассмотрим следующие технологии и направления, которые весьма часто ассоциируют с SDN и упоминают при обсуждении SDN:

- OpenFlow;
- виртуализация сетевых функций (Network Functions Virtualization, NFV);
- виртуальная коммутация;
- виртуализация сети;
- прикладные программные интерфейсы (API) устройств;
- автоматизация сети;
- аппаратная коммутация;
- сетевые фабрики центров данных;
- SD-WAN (программно определяемая глобальная сеть);
- специализированные сетевые контроллеры.

Tlgm: @it_boooks

Мы преднамеренно не даем определение программно определяемой сети (SDN) в этой книге. Несмотря на то что SDN упоминается в текущей главе, главное внимание здесь сосредоточено на основных направлениях и тенденциях, которые часто классифицируются как SDN. Это сделано для того, чтобы читатель узнал как можно больше о каждом из этих направлений.

Из всех перечисленных направлений основное внимание в книге уделено автоматизации сети, прикладным программным интерфейсам (API) и объемлющим (внешним) технологиям, которые чрезвычайно важны для понимания того, каким образом все эти компоненты объединяются в сетевых устройствах, предоставляющих программируемые интерфейсы в совокупности с современными инструментальными средствами автоматизации и управления.

OpenFlow

Протокол OpenFlow уже был представлен читателю, но есть еще несколько важных фактов, которые следует знать.

Одним из самых главных преимуществ использования протокола, подобного OpenFlow, между контроллером и сетевыми устройствами была реальная независимость провайдера (организатора сети) от программного обеспечения контроллера, иногда обозначаемого как сетевая операционная система (network operating system, NOS), и от виртуальных и физических сетевых устройств более низкого уровня. Тем не менее в действительности сетевые провайдеры, использующие OpenFlow в своих решениях (например, Big Switch Networks, HP, NEC), разработали расширения протокола OpenFlow в соответствии с развитием стандартов и из-за необходимости предоставления специфических дополнительных весьма полезных функциональных возможностей, которые базовая серийная версия OpenFlow предложить не может. Но можно предположить, что в конечном итоге все эти расширения будут включены в будущие версии, реализующие стандарт OpenFlow.

При использовании OpenFlow вам предоставляется больший уровень детализации при управлении маршрутами трафика в сети, но бóльшая власть влечет за собой и бóльшую ответственность. Хорошо, если у вас есть группа разработчиков. Например, корпорация Google на основе протокола OpenFlow развернула глобальную сеть B4, которая увеличила производительность почти на 100%. Для большинства других организаций использование OpenFlow или другого подобного протокола будет менее важным, чем комплексные всеобъемлющие решения, ориентированные на поддержку конкретного типа бизнеса.



Несмотря на то что этот раздел называется OpenFlow, с точки зрения архитектуры он посвящен разделению уровня управления и уровня данных. OpenFlow – это основной протокол, используемый для выполнения данной функции.

Виртуализация сетевых функций

Виртуализация сетевых функций (Network Functions Virtualization, NFV) представляет собой не слишком сложную концепцию сетевой архитектуры. По су-

Tlgm: @it_boooks

ществу, она означает развертывание функций, которые ранее обычно выполнялись аппаратурой, в виде программного обеспечения. Самыми известными примерами реализации этой концепции являются виртуальные машины, работающие как маршрутизаторы, сетевые экраны (брандмауэры), балансировщики нагрузки, IDS/IPS, VPN, защитные экраны прикладного уровня (технология WAF) и выполняющие многие другие функции/сервисы.

Появление NFV позволило ослабить казавшуюся незыблемой базовую позицию аппаратуры, стоимость которой могла достигать десятков или даже сотен тысяч долларов с сотнями и тысячами строк команд, и перейти к конфигурированию сети с помощью N компонентов программного обеспечения, то есть с помощью виртуальных устройств. Эти небольшие устройства стали намного более управляемыми, если рассматривать их как отдельные независимые сетевые устройства.



При описанном выше подходе используются виртуальные устройства как форм-фактор для реальных устройств, поддерживающих концепцию виртуализации сетевых функций. Это всего лишь один из примеров. Развертывание сетевых функций как программного обеспечения может происходить во многих различных формах, включая встраивание в гипервизор, в виде контейнера или как приложение, работающее на x86-сервере.

Нередко встречаются случаи развертывания оборудования, потребность в котором с большой вероятностью возникнет в течение будущих трех–пяти лет, так как постепенные обновления слишком сложны и даже являются более дорогостоящими. Таким образом, эксплуатируемую основную стоимость составляет не только оборудование, оно лишь используется для сценариев типа «что, если» для вариантов роста и расширения. Развертывание решений, основанных на программном обеспечении, то есть NFV-решений, предлагает более эффективный способ масштабирования и сведения к минимуму проблемной области критических сбоев всей сети или конкретного приложения при использовании модели «оплата по мере роста» (pay-as-you-grow model). Например, вместо приобретения одного крупного аппаратного межсетевого экрана Cisco ASA вы можете постепенно развертывать виртуальные устройства Cisco ASAv и платить по мере роста. Точно с такой же легкостью можно масштабировать балансировщики нагрузки с помощью новейших технологий, предлагаемых компаниями, подобными Avi Networks.

Если концепция NFV способна предложить столь весомые преимущества, то почему не наблюдается изобилия реально внедренных в эксплуатацию решений и программных продуктов этой категории? По нескольким различным причинам. Во-первых, необходимо полное переосмысление архитектуры сети. При установке единственного монолитного сетевого экрана (как пример) все проходят через этот экран, то есть все приложения и все пользователи, а если не все, то определенная группа, известная системному администратору. В современной модели NFV, где возможно развертывание многочисленных виртуальных сетевых экранов, создается сетевой экран для каждого приложения или

Tlgm: @it_boooks

пользователя в противоположность одному общему экрану, «ответственному за все». Такой подход существенно сужает проблемную область критических сбоев, ограничивая ее одним сетевым экраном или любым другим сервисным устройством, а при необходимости внесения изменений или развертывания нового приложения никоим образом не затрагиваются другие сетевые экраны для прочих приложений (или пользователей).

С другой стороны, в более привычном всем мире монолитных устройств чрезвычайно важно наличие единой панели управления стратегией защиты, обычно в форме интерфейса командной строки (CLI) или графического пользовательского интерфейса (GUI). Возможно, такой подход значительно расширяет потенциальную проблемную область критических сбоев, но предлагает администраторам четко определенную стратегию управления, поскольку управление требуется для единственного устройства. При наличии группы или отдела поддержки таких устройств вполне естественным становится выбор монолитно-ориентированного подхода. Сегодня это действительность, но есть надежда, что через некоторое время с появлением более усовершенствованных инструментов, оказывающих помощь в эксплуатации и управлении решениями, основанными на программном обеспечении, в промышленном масштабе мы увидим более широкое практическое применение этого типа технологии. В мире современных автоматизированных сетевых операций и автоматизированного управления сетями все меньшее значение имеет выбор сетевой архитектуры с точки зрения продуктивности выполнения функций, поскольку имеется возможность намного более эффективного управления как одним устройством, так и большим количеством устройств.

Наряду с вопросами управления другим фактором, влияющим на ситуацию, является то, что многие производители уделяют недостаточно внимания продажам своих версий виртуальных устройств. Мы не утверждаем, что у производителей отсутствуют версии с виртуализацией, но обычно эти версии не являются предпочтительными у многих известных производителей сетевого оборудования. Если поставщик начал свою деятельность по продаже сетевого оборудования в течение нескольких последних лет, то можно заметить его резкий переход к модели, основанной на программном обеспечении, с учетом перспективы продаж и получения прибыли.

Как и во многих технологичных отраслях, одним из главных преимуществ концепции NFV также является ее гибкость и адаптируемость. Исключение аппаратных устройств уменьшает время внедрения новых сервисов, поскольку не требуется время на установку и монтирование оборудования и кабелей, на настройку и подключение к существующей сетевой среде. При использовании программной методики время подключения зависит только от времени развертывания новой виртуальной машины в сетевой среде, а присущие такому подходу преимущества можно получать многократно при клонировании и резервировании виртуальных устройств для будущего тестирования, например в рабочих средах восстановления после катастроф (disaster recovery, DR).

Tlgm: @it_boooks

Наконец, при развертывании NFV исчезает необходимость маршрутизации трафика через специализированное физическое устройство с целью создания требуемого сервиса.

Виртуальная коммутация

На сегодняшний день самыми известными на рынке средствами виртуальной коммутации являются VMware standard switch (VSS), VMware distributed switch (VDS), Cisco Nexus 1000V, Cisco Application Virtual Switch (AVS) и виртуальный коммутатор с открытыми исходными кодами Open vSwitch (OVS).

Эти коммутаторы весьма часто упоминаются в спорах о сущности SDN, но в действительности они представляют собой программные коммутаторы, которые размещены в ядре гипервизора, предоставляя возможность установления соединений в локальной сети между виртуальными машинами (а в последнее время и между контейнерами). Программные коммутаторы предоставляют такие функции, как распознавание MAC-адреса, и такие возможности, как объединение каналов связи, SPAN и sFlow, аналогичные функциям их физических конкурентов, выполняемым в течение многих лет. Несмотря на то что эти виртуальные коммутаторы часто применяются в более крупных решениях SDN и виртуализации сети, они представляют собой лишь коммутаторы, реализованные в форме программного обеспечения. Хотя сами по себе виртуальные коммутаторы не являются законченным решением, они чрезвычайно важны, так как способствуют прогрессу всей отрасли. Виртуальные коммутаторы создали новый уровень доступа или новое направление в деятельности центров данных. На границах сети уже не применяется физический стоечный (top-of-rack, TOR) коммутатор в аппаратном исполнении с ограниченной гибкостью (с точки зрения развития возможностей/функциональности). С тех пор, как границы сетей стали «программными», благодаря использованию виртуальных коммутаторов появилась возможность более быстрого создания новых сетевых функций, реализуемых в программном обеспечении, следовательно, существенно упростилось распространение стратегий управления по сети. Например, стратегия управления защитой может быть развернута на порту виртуального коммутатора, ближайшего к реальному конечному узлу сети. Коммутатор может быть виртуальной машиной или контейнером, предназначенным для дальнейшего расширения стратегии защиты для всей сети.

Виртуализация сети

Решения, классифицируемые как виртуализация сети, уже стали синонимами SDN-решений. В рамках тематики этого раздела виртуализация сети обозначает исключительно программные решения, основанные на оверлейном протоколе. Самыми известными решениями из этой категории являются VMware NSX, Nuage Virtual Service Platform (VSP) и Juniper Contrail.

Главная характеристика этих решений – оверлейный протокол, такой как Virtual eXtensible LAN (VxLAN), используемый для создания соединений между виртуальными коммутаторами на основе гипервизора. Этот тип соединений

Tlgm: @it_boooks

и методика туннелирования (tunneling) обеспечивает совместимость на уровне канала передачи (Layer 2) между виртуальными машинами, существующими на различных независимых физических хостах физической сети, при этом подразумевается, что физическая сеть может быть сетью уровня 2 (Layer 2), уровня 3 (Layer 3) или комбинацией обоих уровней. В результате создается отделенная от физической сети виртуальная сеть, которая предоставляет свободу выбора и гибкость.



Следует отметить, что термин оверлейная сеть (overlay network) часто используется вместе с термином андерлейная (нижележащая) сеть (underlay network). Поэтому следует уточнить, что андерлейная означает нижележащая физическая сеть, созданная с помощью физических кабелей. Оверлейная сеть формируется с использованием решения виртуализации сети, которое динамически создает туннели (tunnels) между виртуальными коммутаторами в пределах центра данных. И снова отметим, что это следует рассматривать в контексте решения виртуализации сети на основе программного обеспечения. Также отметим, что многие решения на аппаратной основе в настоящее время развертываются с помощью VxLAN в качестве оверлейного протокола для создания туннелей уровня 2 (Layer 2, канальный) между стоечными (top-of-rack) устройствами в пределах центра данных уровня 3 (Layer 3, сетевой).

Несмотря на то что оверлейный протокол является лишь элементом реализации решения виртуализации сети, эти решения представляют собой гораздо большее, нежели простой набор виртуальных коммутаторов, объединяемых оверлеями. Обычно это полные, всеобъемлющие решения, предлагающие функции защиты, балансировки нагрузки и обратную совместимость при взаимодействии с физической сетью, имеющей единственный пункт управления (например, контроллер). Часто такие решения также обеспечивают объединение с сервисами лучших компаний в сфере сетевых услуг уровней 4–7 (Layer 4–7), предлагая на выбор ряд технологий, которые можно развернуть на платформах виртуализации сети.

Кроме того, гибкость (адаптируемость) достигается благодаря наличию центральной управляющей платформы, которая используется для динамического конфигурирования каждого виртуального коммутатора и необходимых сервисных компонентов. Напомним, что застой в развитии эксплуатационных характеристик сетей произошел из-за вездесущего преобладания интерфейса командной строки (CLI), предпочитаемого всеми производителями в реальном рыночном мире. При виртуализации сети нет необходимости вручную конфигурировать все виртуальные коммутаторы, так как любое решение упрощает этот процесс, предоставляя централизованный интерфейс (GUI, CLI), а также прикладной программный интерфейс (API), с помощью которого все изменения можно реализовать программно.

Прикладные программные интерфейсы сетевых устройств

В течение нескольких последних лет производители начали понимать, что одного лишь стандартного интерфейса командной строки уже недостаточно

Tlgm: @it_boooks

и его использование ограничивает функциональность. Если вы когда-либо работали с любым языком программирования или скриптовым языком, то, вероятнее всего, сможете понять эту проблему. В любом случае, мы более подробно рассмотрим эту тему в главе 7.

Главным проблемным пунктом является то, что скрипты для старых и современных сетевых устройств, использующих только интерфейс командной строки, не возвращают структурированные данные. То есть предполагается, что данные передаются из устройства в скрипт в простейшем текстовом формате (например, вывод команды `show version`), затем требуется написание отдельного скрипта для синтаксического разбора этого текста, чтобы извлечь необходимые атрибуты, такие как время непрерывной работы или версию операционной системы. Даже при незначительных изменениях вывода команд `show` скрипты могут стать неработоспособными из-за неверных правил синтаксического разбора. Хотя такой подход применялся и продолжает применяться всеми администраторами, с технической точки зрения автоматизация стала возможной уже давно, но только сейчас производители постепенно переходят на сетевые устройства, управляемые прикладными программными интерфейсами (API).

Предоставление API исключает необходимость синтаксического разбора неструктурированного текста, так как сетевое устройство возвращает структурированные данные, существенно сокращая время, затрачиваемое на написание скрипта. Вместо парсинга «сырого» текста в поисках времени непрерывной работы или любого другого атрибута возвращается объект, предоставляющий именно ту информацию, которая необходима в данном конкретном случае. При этом не только сокращается время написания скрипта и снижается «входной» уровень требований для сетевых инженеров (и прочих непрограммистов), но также предоставляется более четко определенный интерфейс, с помощью которого профессиональные разработчики программного обеспечения могут быстро разрабатывать и тестировать код почти так же, как они используют API для несетевых устройств. «Тестирование кода» может означать проверку (испытания) новых топологий, сертификацию новых сетевых функциональных возможностей, валидацию конкретных сетевых конфигураций и многое другое. Все эти операции сегодня выполняются вручную, на них затрачивается очень много времени, при этом весьма высока вероятность возникновения ошибок.

Один из первых широко распространенных API-интерфейсов в сетевой сфере представила компания Arista Networks. Ее интерфейс называется eAPI и представляет собой прикладной программный интерфейс на основе протокола HTTP, который использует данные в JSON-кодировке. Разумеется, API на основе протокола HTTP и использование формата JSON будут подробно рассматриваться начиная с главы 5. После Arista компания Cisco представила свои API-интерфейсы, такие как Nexus NX-API и NETCONF/RESTCONF, на конкретных платформах, и появились производители и поставщики, такие как

Tlgm: @it_boooks

Juniper, получившие в свое полное распоряжение расширяемый и адаптируемый интерфейс NETCONF, но это не привлекло особого внимания. Следует отметить, что в те времена почти каждый производитель/поставщик предлагал собственный тип API.

Более подробно эта тема будет рассматриваться в главе 7.

Автоматизация сети

По мере развития прикладных программных интерфейсов в сетевой отрасли продолжают появляться все более интересные варианты извлечения преимуществ из их использования. В ближайшем будущем автоматизация сети является главным потенциальным средством получения преимуществ программируемых интерфейсов при использовании современных сетевых устройств, предоставляющих API.

В более широком смысле автоматизация сети – это не только автоматизация процедур конфигурирования сетевых устройств. Разумеется, это самое распространенное представление об автоматизации сети, но использование API и программируемых интерфейсов обеспечивает гораздо бóльшие потенциальные возможности автоматизации, нежели простое манипулирование параметрами конфигурации.

Переход к использованию API делает простым и ясным доступ ко всем данным, скрытым в сетевых устройствах. Здесь имеются в виду данные уровня информационных потоков, таблицы маршрутизации, информационные базы данных переадресации (FIB – Forwarding Information Base), статистические данные интерфейсов, таблицы MAC-адресов, таблицы виртуальных локальных сетей (VLAN), серийные номера устройств – этот список можно продолжать бесконечно. Использование современных методик автоматизации, которые, в свою очередь, применяют API, может оказать оперативную помощь в выполнении повседневных операций управления сетью, сбора данных и автоматизированной диагностики. Кроме того, поскольку современные API позволяют извлекать структурированные данные, администраторы получают возможность выводить и анализировать именно те наборы данных, которые необходимы, и даже объединять выходные данные от нескольких различных команд `show`, значительно сокращая время на отладку и устранение проблем в сети. Вместо установления соединений с N-маршрутизаторами, работающими по протоколу граничного шлюза (BGP – Border Gateway Protocol), с целью проверки правильности конфигурации или устранения возникшей проблемы можно воспользоваться методиками автоматизации для упрощения этого процесса.

Кроме того, применение методик автоматизации в конечном итоге позволяет создать более предсказуемую и единообразную сеть в целом. Это можно наблюдать при автоматизации создания файлов конфигурации, при автоматизации создания виртуальной локальной сети (VLAN) и/или при автоматизации процесса устранения проблем и неисправностей. Это упрощает понимание обобщенного процесса всеми пользователями, поддерживающими конкретную сетевую среду, в отличие от ситуации, в которой каждый сетевой адми-

Tlgm: @it_boooks

нистратор должен формировать свою собственную индивидуальную практическую методику работы.

Различные типы автоматизации сети подробно рассматриваются в главе 2.

Аппаратная коммутация

Аппаратную коммутацию (bare-metal switching) тоже часто приравнивают к SDN, но в действительности это ошибочное представление. Ранее уже было отмечено, что в этой главе даются краткие описания различных технологических направлений, которые воспринимаются как SDN, следовательно, и этой теме нужно уделить немного внимания. Если вернуться в 2014 год (и даже немного раньше), то можно обнаружить, что для обозначения аппаратной коммутации в то время использовался термин «коммутация с помощью прозрачного ящика» (white-box switching) или commodity switching. С тех пор термин изменился, и не без серьезных на то оснований.

Прежде чем рассматривать замену «прозрачного ящика» (white-box) на «голую» аппаратуру (bare-metal), необходимо понять, что это означает на самом верхнем уровне с учетом существенных изменений в представлениях о том, что представляют собой сетевые устройства. В последние 20 лет сетевые устройства всегда приобретались как физические устройства, точнее, как аппаратные устройства, сетевая операционная система и инструменты/приложения, которые можно было использовать в сетевой операционной системе. Все эти компоненты приобретались у одного поставщика/производителя.

В соответствии с концепцией прозрачного ящика и «чисто» аппаратных сетевых устройств такое устройство по большей части похоже на x86-сервер (см. рис. 1.3). Это позволяет пользователю отделить друг от друга все требуемые компоненты, предоставляя возможность купить аппаратуру у одного производителя, операционную систему у другого производителя, а затем подбирать и загружать инструменты/приложения от различных поставщиков или даже воспользоваться программным обеспечением с открытыми исходными кодами.

Коммутация по методике прозрачного ящика стала весьма популярной в период повышенного интереса к OpenFlow, поскольку целью этой методики являлось превращение аппаратуры в товар массового потребления и сосредоточение «интеллекта» сети в OpenFlow-контроллере, который сейчас называют SDN-контроллером. В 2013 году корпорация Google объявила о создании собственных коммутаторов под управлением OpenFlow. Тогда это объявление стало предметом многочисленных дискуссий между профессионалами, но в действительности далеко не каждого конечного пользователя можно сравнивать с Google и не каждый пользователь будет создавать собственную программную или аппаратную платформу.

Вместе с этими событиями наблюдалось появление нескольких компаний, сосредоточивших свои усилия на предоставлении решений на основе коммутации по методике прозрачного ящика, например Big Switch Networks, Cumulus Networks и Pica8. Каждая такая компания предлагает решения исключительно

Tlgm: @it_boooks

на основе программного обеспечения, но при этом остается необходимость в аппаратуре, на которой должно работать программное обеспечение, чтобы предоставить полноценное завершенное решение. Первоначальным источником этих аппаратных платформ «прозрачного ящика» были ODM-компании[1], такие как Quanta, Super Micro и Accton. Даже если вы работали в сетевой отрасли, вряд ли вам что-либо известно об этих компаниях.

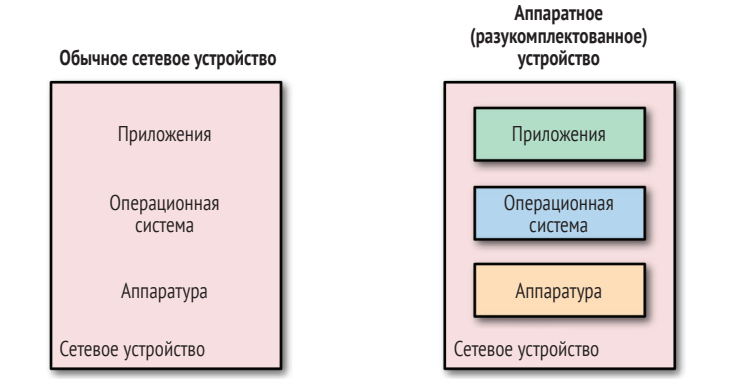


Рис. 1.3 ❖ Общая схема обычного стека коммутации и стека аппаратной коммутации

До тех пор, пока компании Cumulus и Big Switch не объявили о сотрудничестве с крупными корпорациями, в том числе с HP и Dell, не изменялось и название этого направления – с «прозрачного ящика» на «аппаратную коммутацию», но теперь крупные известные производители поддерживали операционные системы от других компаний, таких как Big Switch и Cumulus Networks, на своих аппаратных платформах.

Возможно, до сих пор остается недопонимание того факта, что аппаратная коммутация с технической точки зрения не является SDN, поскольку производители, подобные Big Switch, действуют в обоих направлениях. На возникающие в связи с этим вопросы можно дать простой ответ. Если контроллер объединен с программным решением, использующим протокол типа OpenFlow (но не обязательно OpenFlow), а обмен данными между сетевыми устройствами организован с помощью программного обеспечения, то все это в совокупности очень похоже на программно определяемую сеть (SDN). Именно так действует компания Big Switch – загружает программное обеспечение на «голую» аппаратуру (прозрачный ящик), затем инициализирует работу агента OpenFlow, обеспечивающего обмен данными с контроллером как компонентом этого решения.

[1] ODM – Original Design Manufacturers.

Tlgm: @it_boooks

С другой стороны, компания Cumulus Networks предлагает специализированный дистрибутив Linux, ориентированный на работу с сетевыми коммутаторами. Этот дистрибутив, или операционная система, поддерживает реализации обычных стандартных протоколов, таких как LLDP, OSPF и BGP, без каких-либо конкретных требований к контроллеру, то есть обеспечивает сопоставимость и совместимость с сетевыми архитектурами, которые не основаны на SDN.

Приведенное выше описание должно стать свидетельством того, что Cumulus является компанией, создающей сетевые операционные системы и обеспечивающей работу своего программного обеспечения на чисто аппаратных коммутаторах, в то время как Big Switch – компания, создающая SDN на аппаратной основе, где требуется использование SDN-контроллера этой компании, но при этом Big Switch поддерживает инфраструктуру аппаратной коммутации других производителей.

Если подвести краткий итог, то аппаратную коммутацию (коммутатор как прозрачный ящик) можно приблизительно определить как разукомплектацию (разделение на отдельные компоненты) и предоставление возможности приобретения сетевой аппаратуры у одного производителя, а программного обеспечения у другого, право выбора остается за пользователем. В этом случае администраторы получают полную свободу при выборе проектных решений, архитектур и программного обеспечения без необходимости замены аппаратуры; иногда требуется заменить только основную операционную систему.

Сетевые фабрики центров данных

Возникала ли у вас ситуация, когда невозможно обеспечить простое взаимодействие между различными сетевыми устройствами, даже если все они работают по стандартным протоколам, таким как Spanning Tree или OSPF? Если эта проблема вам знакома, то вы не одиноки. Представьте себе сеть центра данных с компактным ядром и отдельными коммутаторами в каждой аппаратной стойке. А теперь подумайте, какие действия необходимо выполнить, когда наступает время обновления.

Существует много способов обновления сетей, подобных вышеупомянутой, но что, если надо обновить только стоечные (top-of-rack, TOR) коммутаторы и при исследовании текущего рынка TOR-коммутаторов был выбран новый производитель или новая платформа? Это вполне обычная ситуация, она возникает достаточно часто. Сам процесс прост – обеспечение взаимодействия между новыми коммутаторами и существующим ядром (разумеется, мы предполагаем, что в ядре имеются доступные порты) и правильное конфигурирование магистрального канала связи по стандарту 802.1Q, если это соединение уровня 2 (канальный уровень), или конфигурирование предпочитаемого протокола маршрутизации, если это соединение уровня 3 (сетевой уровень).

Здесь начинают действовать сетевые фабрики центров данных (data center network fabrics). Именно они призваны полностью изменить представление о работе сетей центров данных.

Сетевые фабрики центров данных ориентированы на изменение образа мышления сетевых операторов: от управления отдельными устройствами

Tlgm: @it_boooks

поочередно к управлению всей системой как единым целым. Если вернуться к описанному в начале раздела сценарию, то здесь вряд ли появится возможность замены на TOR-коммутатор от другого производителя, так как коммутатор является всего лишь отдельным компонентом сети центра данных. Но когда сеть развертывается и управляется как единая система, необходимо воспринимать ее как единую систему. Это означает, что процесс обновления должен представлять собой переход от системы к системе, от фабрики к фабрике. В таком контексте фабрики могут полностью заменяться во время процедур обновления, но не допускается замена отдельных компонентов внутри фабрики, по крайней мере, в большинстве случаев. Такой подход возможен, если конкретный производитель/поставщик предоставляет подробный план (маршрут) перехода или обновления и только при использовании аппаратной коммутации (то есть заменяется только аппаратура). Примерами сетевых фабрик центров данных являются Cisco Application Centric Infrastructure (ACI), Big Switch Big Cloud Fabric (BCF) или фабрика и гиперконвергентная сеть (hyperconverged network) компании Plexxi (https://www.plexxi.com/).

В дополнение к интерпретации сети как единой системы сетевые фабрики центров данных также обладают следующими общими характеристиками:

- единый интерфейс для управления и конфигурирования всей фабрики, включая управление стратегиями;
- распределенные по всей фабрике шлюзы, определяемые по умолчанию;
- возможности определения многих различных маршрутов;
- использование одной из форм SDN-контроллера для управления системой.

SD-WAN

В последние два года одним из самых интенсивно разрабатываемых направлений SDN стала методика программно определяемой глобальной сети (Software Defined Wide Area Networking, SD-WAN). За последние годы выросло количество компаний, всерьез занявшихся решением проблем, связанных с глобальными сетями (WAN), в их числе Viptela (совсем недавно поглощенная корпорацией Cisco), CloudGenix, VeloCloud, Cisco IWAN, Glue Networks и Silverpeak.

В сфере глобальных сетей не наблюдалось значительных технологических прорывов со времени перехода с протокола ретрансляции кадров Frame Relay на механизм коммутации MPLS. С появлением широкополосных каналов связи и интернета затраты становятся сопоставимыми с затратами на аналогичные частные линии связи, со временем развитие продолжается с применением VPN-туннелей, связывающих сайты, в итоге все это закладывается в основу следующей важной вехи – глобальной сети (WAN).

Часто применяемые проектные решения для удаленных офисов обычно включают частную линию связи (с универсальным многопротокольным механизмом передачи данных – multiprotocol label switching, MPLS) и/или открытое интернет-соединение. При наличии обоих вариантов интернет обычно используется только как резервный вариант, главным образом для входящего («гостевого») трафика или для данных общего пользования, передаваемых

Tlgm: @it_boooks

через VPN в корпоративную систему, тогда как MPLS-канал используется для приложений, требующих минимальных задержек, таких как голосовая связь или канал видеосвязи. Когда трафик начинает разделяться между каналами связи, увеличивается сложность конфигурирования протокола маршрутизации, а также ограничивается детализация при определении маршрутов к целевому адресу. Адрес источника, приложение и производительность (пропускная способность) сети в режиме реального времени, как правило, не принимаются во внимание в решениях по определению наилучшего маршрута.

Общая архитектура программно определяемой глобальной сети (SD-WAN), используемая во многих современных решениях, похожа на решение по виртуализации сети, используемое в центре данных, тем, что здесь также применяется оверлейный протокол для взаимодействия граничных устройств SD-WAN. Поскольку задействован оверлейный протокол, в таком решении не имеет значения, какая именно физическая среда передачи данных более низкого уровня используется в данном конкретном случае, что обеспечивает функционирование SD-WAN через интернет или поверх частной глобальной сети (WAN). Эти решения нередко работают на основе двух и более каналов связи интернета на вспомогательных сайтах с полным зашифрованием трафика с использованием IPSec. Кроме того, многие из этих решений постоянно измеряют производительность каждого канала связи, чтобы при перегрузках быстро переключаться между каналами, обеспечивая работу важных приложений даже в самые сложные периоды. Так как здесь прикладной уровень зримо участвует в процессе, администраторы также могут выбирать приложения, для которых должен определяться конкретный маршрут. Подобного рода функциональные возможности не часто обеспечиваются в архитектурах глобальных сетей, которые полагаются исключительно на маршрутизацию по предопределенному целевому пункту с использованием обычных протоколов маршрутизации, таких как OSPF и BGP.

С точки зрения архитектуры решения SD-WAN от упоминаемых выше производителей, таких как Cisco, Viptela и CloudGenix, также обычно предлагают некоторую форму автоматической установки и конфигурирования (zero-touch provisioning, ZTP) и централизованного управления с помощью портала на стороне клиента или в облачной среде в виде SaaS-приложения, существенно упрощающего управление и эксплуатацию развивающиейся глобальной сети.

Полезный побочный эффект от использования технологии SD-WAN проявляется в том, что эта технология предлагает конечным пользователям бóльшую свободу выбора, так как в глобальной сети, как и в интернете, можно использовать любое устройство передачи данных и любой тип соединения. При этом упрощается конфигурирование и снижается сложность основных магистральных сетей, что, в свою очередь, позволяет упростить их внутреннюю организацию и архитектуру и, возможно, сократить расходы на их поддержку. Если заглянуть чуть дальше с технической точки зрения, то можно утверждать, что все конструкции логических сетей, таких как Virtual Routing and Forwarding (VRF), должны управляться через пользовательский интерфейс (user interface, UI) на

Tlgm: @it_boooks

платформе контроллера, который предоставляет производитель SD-WAN. При этом опять же исчезает необходимость ожидания в течение нескольких недель ответной реакции на затребованные пользователем изменения.

Специализированные сетевые контроллеры

Возможно, вы уже заметили, что некоторые из описанных выше направлений и тенденций взаимосвязаны в той или иной степени. Это один из моментов, сложных для понимания любой новой технологии и любого нового направления развития техники в последние несколько лет.

Например, широко применяемые платформы виртуализации сетей используют контроллер как средство нескольких вариантов решений, которые также попадают в категории сетевых фабрик центров данных, SD-WAN и аппаратной коммутации. Вы в замешательстве? Возможно, у вас возникает вопрос: почему сеть на основе контроллеров сама по себе не выделена в отдельную категорию? В действительности чаще всего это лишь характеристика и механизм реализации современных решений, но не все описанные выше направления в полной мере охватывают функциональные возможности, которые могут предоставить контроллеры с технической точки зрения.

Например, широко известный SDN-контроллер OpenDaylight (ODL), показанный на рис. 1.4, как и многие другие контроллеры, представляет собой платформу, а не рыночный продукт. Это платформы, которые могут предлагать специализированные приложения, например виртуализацию сети, но, кроме

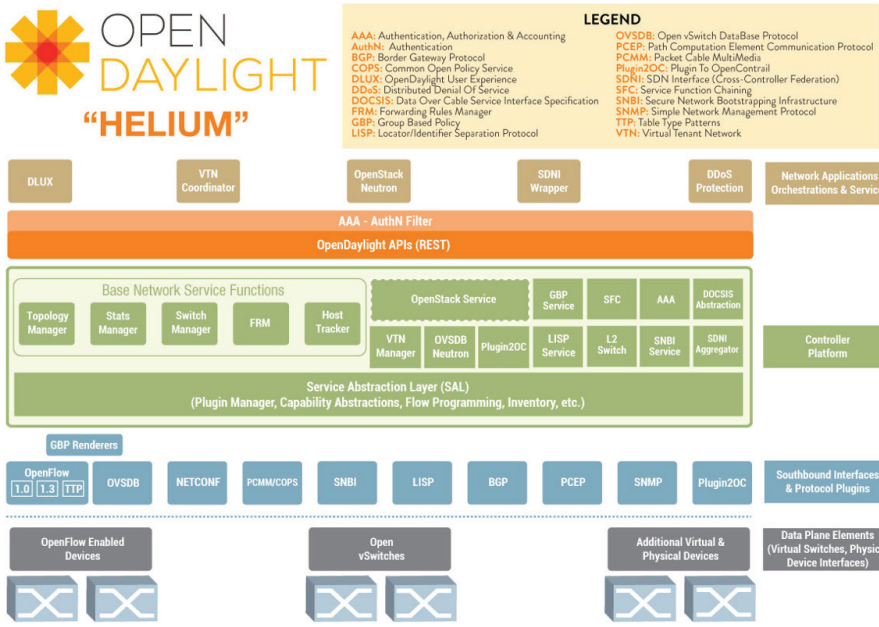


Рис. 1.4 ❖ Архитектура контроллера (платформы) OpenDaylight

Tlgm: @it_boooks

того, эти платформы используются для мониторинга и наблюдения за состоянием сетей, для компоновки и укрупнения сетей и для выполнения многих других функций в сочетании с приложениями, работающими поверх платформы контроллера. Именно поэтому очень важно понять, что могут предложить контроллеры сверх того, что предлагают более привычные приложения, такие как фабрики, средства виртуализации сети и SD-WAN.

Резюме

Вы получили общее представление о направлениях и технологиях, которые чаще всего относят к категории программно определяемых сетей, указывающих путь к повышению эффективности операций, выполняемых в сети, с помощью автоматизации и реализации программируемости сети. За последние несколько лет были созданы десятки SDN-стартапов, в них инвестированы миллионы в форме венчурных капиталов, и миллиарды потрачены на приобретение (поглощение) этих компаний. Может показаться невероятным, но если заглянуть чуть дальше в будущее, то все это делалось и делается с одной общей целью – реализовать программные принципы и технологии, чтобы предложить пользователям бóльшую мощь, управляемость, гибкость и свободу выбора применяемой технологии с одновременным повышением эффективности эксплуатации сетей.

В главе 2 будет рассматриваться автоматизация сети, будут более подробно описаны различные типы автоматизации, некоторые основные протоколы и прикладные программные интерфейсы, а также успехи в развитии автоматизации сетей за последние несколько лет.

Tlgm: @it_boooks

Глава 2

Автоматизация сети

В этой главе основное внимание уделено рассмотрению основных концепций автоматизации сети на высоком уровне, с тем чтобы лучше подготовить читателя к пониманию содержимого каждой следующей главы в отдельности и книги в целом.

Для достижения указанной цели в эту главу включены следующие разделы:

- для чего нужна автоматизация сети – рассматриваются различные обоснования применения автоматизации и повышение эффективности сетевых операций как доказательство того, что автоматизация представляет собой нечто большее, чем сокращение времени создания конфигураций для сетевых устройств;
- типы автоматизации сети – рассматриваются различные типы автоматизации: от обычного управления конфигурациями до автоматизации диагностики сети и устранения возникающих проблем, что опять же подтверждает тот факт, что автоматизация – это не только более быстрое внесение изменений;
- развитие уровня управления от протокола SNMP до прикладных программных интерфейсов устройств – краткое введение и описание нескольких различных типов API для сетевых устройств в прошлом и в настоящем;
- автоматизация сети в эпоху SDN – краткий обзор причин, по которым средства автоматизации сети остаются чрезвычайно важными в применении к программно определяемой сети, при этом особое внимание уделено архитектурам на основе контроллеров и соответствующим развертываемым решениям.

Данная глава не является сложным техническим текстом, она дает лишь общее представление об идеях и концепциях автоматизации сети. Здесь описываются общие принципы и закладывается основа для понимания следующих глав.

Для чего нужна автоматизация сети

Автоматизация сети, как и многие типы автоматизации, определяется как набор средств для более быстрого выполнения рабочих операций. Повышение

Tlgm: @it_boooks

производительности работы – это прекрасно, но сокращение времени развертывания и внесения изменений в конфигурацию далеко не всегда является именно той задачей, которую необходимо решить многим ИТ-организациям.

Не забывая о скорости, рассмотрим несколько причин, по которым ИТ-организации всех типов и размеров должны постепенно переходить к использованию сетевой автоматизации. Следует отметить, что те же самые принципы применяются и в других типах автоматизации (для прикладного ПО, для систем в целом, для хранилищ данных, для телефонии и т. д.).

Упрощение архитектуры

В настоящее время конфигурации очень многих сетевых устройств настолько отличаются друг от друга, что их можно сравнить с неповторяющимися формами снежинок, а сетевые инженеры испытывают чувство гордости за предлагаемые ими решения задач по организации доставки данных и обеспечения работы приложений с помощью одноразовых специфических изменений, которые в итоге не только осложняют обслуживание и управление сетью, но и затрудняют ее автоматизацию.

Автоматизацию и управление сетью не следует рассматривать как второстепенный проект или «необязательное дополнение», эти вопросы должны быть решены на самом раннем этапе проектирования новых архитектур. Кроме того, это решение должно содержать соответствующим образом сформированный бюджет для персонала и комплекта инструментальных средств. К сожалению, комплект инструментальных средств почти всегда становится первым пунктом в списке кандидатов на удаление, когда бюджет сокращается.

Обобщенная архитектура и соответствующие операции «второго дня» (day 2 operation; обычные рабочие операции, выполняемые после установки и настройки системы) должны быть одними и теми же. При проектировании архитектуры необходимо тщательно продумать ответы на следующие вопросы:

- какие функциональные возможности поддерживаются всеми производителями?
- какие расширения работают независимо от конкретной платформы?
- какой тип API или комплект инструментов автоматизации работает с конкретными платформами сетевых устройств?
- имеется ли подробная документация на API?
- какие библиотеки существуют для рассматриваемого продукта?

Если ответы на эти вопросы получены на самом раннем этапе проектирования, то итоговая архитектура будет более понятной и управляемой, легко воспроизводимой и более простой для сопровождения и автоматизации, с наличием в эксплуатируемой сети минимального количества расширений, являющихся охраняемой собственностью производителя.

Даже после упрощения архитектуры и развертывания ее с обеспечением правильного управления и наличия комплекта инструментов автоматизации следует всегда помнить о необходимости сведения к минимуму одноразовых

Tlgm: @it_boooks

специфических изменений, чтобы конфигурация сети снова не стала напоминать снежинку уникальной формы.

Детерминированные результаты

В крупной производственной организации обычным делом являются коллективные обзоры изменений, целью которых являются рассмотрение предстоящих изменений в сети, оценка их воздействия на внешние системы и составление планов возврата к предыдущему состоянию (rollback plan). При использовании интерфейса командной строки для внесения таких изменений последствия любой опечатки или неправильно введенной команды становятся катастрофическими. Представьте себе группу из 3, 4, 5 или 50 инженеров. У каждого инженера может существовать собственный способ внесения запланированных изменений в подобных ситуациях. Более того, применение интерфейса командной строки и даже графического пользовательского интерфейса не исключает и не снижает вероятность совершения ошибок в процессе внесения изменений.

Использование надежной, тщательно протестированной методики автоматизации сети для внесения изменений помогает обеспечить более предсказуемое поведение, по сравнению с выполнением этой операции вручную, и предоставляет инженерной группе больше шансов на получение детерминированных результатов. Это еще один шаг к правильному и точному выполнению поставленной задачи с полным устранением человеческих ошибок. Решаемая задача может быть любой: от изменений в виртуальной локальной сети (VLAN) до регистрации нового клиента, требующей нескольких изменений во всей сети.

Гибкость бизнеса

Известно, что автоматизация сети обеспечивает скорость и гибкость при внесении изменений, но те же преимущества обеспечены и при извлечении данных из сетевых устройств, выполняя эту операцию так быстро, как требует бизнес, или, с более практической точки зрения, со скоростью, необходимой для оперативного устранения проблем в сети.

После появления виртуализации серверов у администраторов появилась возможность практически мгновенного развертывания новых приложений. Но чем быстрее развертываются приложения, тем больше возникает вопросов о том, почему так долго конфигурируются сетевые ресурсы, такие как виртуальные локальные сети (VLAN), таблицы маршрутов, стратегии сетевых экранов (firewalls), стратегии балансировки нагрузки или все это вместе взятое при развертывании нового приложения от независимого производителя.

Кажется вполне очевидным тот факт, что, применяя автоматизацию сети, сетевые инженеры и группы сетевого сопровождения могут действовать быстрее, чем аналогичные ИТ-специалисты и группы при развертывании приложений, но более важно, что это помогает придать бизнесу бóльшую гибкость. При

Tlgm: @it_boooks

решении вопросов адаптации чрезвычайно важно понимать существующие, часто выполняемые вручную рабочие процессы, прежде чем пытаться применять автоматизацию любого типа, вне зависимости от того, насколько хороши по замыслу намерения сделать бизнес более гибким.

Если вы не знаете, что именно хотите автоматизировать, то процесс автоматизации будет сложным и очень долгим. Самый первый и самый важный совет: начиная автоматизацию сети, всегда достигайте полного понимания существующих рабочих процессов, выполняемых вручную, документируйте их, изучайте их воздействие на бизнес. Тогда процесс развертывания технологии автоматизации и соответствующих инструментальных средств станет намного более простым.

В этом разделе были представлены некоторые обобщенные задачи высокого уровня – от формирования упрощенных архитектур до обеспечения гибкости бизнеса, – с точки зрения которых следует рассматривать автоматизацию сети. В следующем разделе описываются различные типы автоматизации сети.

Типы автоматизации сети

Автоматизация в общем смысле ассоциируется с увеличением скорости работы, и считается, что для некоторых сетевых задач увеличение скорости не требуется, поэтому нетрудно понять, почему некоторые ИТ-группы не оценивают по достоинству значение автоматизации. Ярким примером является конфигурирование виртуальной локальной сети (VLAN), поскольку при этом возникают следующие вопросы: «Насколько быстро должна создаваться VLAN? Сколько сетей VLAN будет добавляться ежедневно? Действительно ли для этого необходима автоматизация?» Это вполне закономерные вопросы.

В этом разделе все внимание будет сосредоточено на некоторых задачах, для которых автоматизация имеет реальный смысл: подготовка и настройка устройств, сбор данных, устранение проблем, создание отчетов и совместимость. При этом следует помнить, что автоматизация – это не просто скорость и гибкость, автоматизация предлагает специалистам, группам, бизнес-предприятиям более предсказуемые и более детерминированные результаты.

Подготовка и настройка устройств

Один из самых простых и быстрых способов приступить к автоматизации сети – начать с автоматизации создания файлов конфигурации устройств. Файлы конфигурации используются для первоначальной подготовки и настройки устройств (device provisioning) и передаются в соответствующие сетевые устройства.

Этот процесс можно разделить на два этапа: создание файла конфигурации, затем передача конфигурации в устройство.

Для автоматизации процесса создания файлов конфигурации сначала необходимо отделить входные данные (inputs), то есть параметры конфигурации,

Tlgm: @it_boooks

от более низкого уровня синтаксиса (командной строки) конфигурации, зависящего от конкретного производителя. В результате будут созданы отдельные файлы со значениями параметров конфигурации, такими как характеристики VLAN, информация о домене, об интерфейсах, о маршрутизации и о прочих подробностях конфигурации, а в конечном итоге – шаблон конфигурации. Более подробно этот процесс будет описан в главе 6.

Сейчас мы будем считать шаблон конфигурации аналогом стандартного универсального шаблона, используемого для всех развертываемых устройств. Применяя методику создания шаблонов сетевой конфигурации (network configuration templating), вы получаете возможность быстро формировать взаимно согласованные файлы конфигурации для конкретной сети. Это означает, что вам больше не понадобится Блокнот для копирования и вставки элементов конфигурации из файла в файл, – разве это не оправдывает затраты времени на автоматизацию?

Два инструментальных средства, существенно упрощающих применение шаблонов конфигурации с переменными (входными данными), – Ansible и Salt. Эти инструменты способны буквально за несколько секунд сгенерировать сотни файлов конфигурации вполне предсказуемым и надежным образом.



Подготовка и генерация файлов конфигурации из шаблонов во всех подробностях рассматривается в главе 6, а реализация процесса создания шаблонов с помощью Ansible и Salt – в главе 9. В текущем разделе показан всего лишь простейший пример верхнего уровня автоматизации.

Рассмотрим пример, в котором берется текущая конфигурация и выполняется ее разделение на шаблон и отдельный файл переменных (входных данных) для более ясного понимания рассматриваемой темы.

Ниже в качестве примера приведен фрагмент файла конфигурации:

```
hostname leaf1
ip domain-name ntc.com
!
vlan 10
   name web
!
vlan 20
   name app
!
vlan 30
   name db
!
```

Если отделить данные от команд, выполняемых в интерфейсе командной строки, то файл будет преобразован в два файла: шаблон и файл данных (значений переменных).

Сначала рассмотрим файл переменных, описанных на языке YAML (язык YAML будет подробно описан в главе 5):

Tlgm: @it_boooks


```
---
hostname leaf1
domain_name: ntc.com
vlans:
  - id: 10
    name: web
  - id: 20
    name: app
  - id: 30
    name: db
```

Еще раз отметим, что в YAML-файле содержатся только данные.



Для этого примера используется язык описания шаблонов Jinja, основанный на языке программирования Python. Язык шаблонов Jinja подробно рассматривается в главе 6.

Итоговый шаблон, предназначенный для работы с показанным выше файлом данных, размещен в файле *leaf.j2* и выглядит следующим образом:

```
!
hostname {{ inventory_hostname }}
ip domain-name {{ domain_name }}
!
!
{% for vlan in vlans %}
vlan {{ vlan.id }}
  name {{ vlan.name }}
{% endfor %}
!
```

В этом примере двойные фигурные скобки обозначают переменную языка шаблонов Jinja. Другими словами, это позиции, в которые подставляются конкретные значения, когда шаблону требуются данные. Поскольку двойные фигурные скобки обозначают переменные, а в шаблоне нет конкретных значений для этих переменных, все подставляемые данные необходимо где-то хранить. В приведенном выше примере мы сохранили их в YAML-файле. Помимо обычных простейших YAML-файлов, можно также использовать скрипт для извлечения этого типа информации из внешней системы, такой как система управления сетью (network management system, NMS) или система управления IP-адресами (IP address management system, IPAM).

В приведенном выше примере, если группе управления VLAN потребуется добавить еще одну виртуальную локальную сеть поверх сетевых устройств, не возникает никаких затруднений. Нужно просто внести соответствующие изменения в файл значений переменных и заново сгенерировать новый файл конфигурации с помощью Ansible или любого другого механизма генерации (Salt, чистый Python и т. д.).



В главе 6 будет рассматриваться применение языка программирования Python к шаблонам Jinja с конкретным примером создания на языке Python скрипта, который можно использовать как простой механизм генерации.

Tlgm: @it_boooks

После того как конфигурация сгенерирована, необходимо передать (push; «протолкнуть») ее в сетевое устройство. Процессы передачи (push) и выполнения (execution) здесь не рассматриваются, так как существует огромное количество способов сделать это, в том числе ряд проприетарных решений по автоматической установке и конфигурированию (zero-touch provisioning, ZTP) различных производителей, а также некоторые другие методики, которые мы рассмотрим в главах 7 и 9.

Кроме того, это был всего лишь начальный уровень знакомства с шаблонами, поэтому если даже вы поняли не все, то не стоит волноваться. Как уже было отмечено выше, работа с шаблонами во всех подробностях описывается в главе 6.

Возможно, более важной задачей, чем создание конфигураций и передача их в устройства, является сбор данных, и именно об этом пойдет речь в следующем разделе.

Сбор данных

Инструментальные средства мониторинга обычно используют протокол SNMP (Simple Network Management Protocol), то есть опрашивают определенные базы данных с управляющей информацией (management information base, MIB) и возвращают извлеченные данные в инструмент мониторинга. Полученные данные в большей или меньшей степени могут представлять ту информацию, которая вам действительно необходима. А что, если запрашиваются статистические данные сетевого интерфейса? Вы можете получить каждый счетчик, который выводится командой `show interface`, но необходима только статистика по сбросам интерфейса (interface resets) и совершенно не нужны данные CRC errors, jumbo frames, output errors и т. п. А если необходимо наблюдать сбросы конкретного интерфейса в корреляции с интерфейсами, соседствующими с устройствами, определенными по протоколам CDP/LLDP, и эти данные нужны сию же секунду, а не в следующем цикле опроса? Как может помочь автоматизация сети в этом случае?

Поскольку главная задача – предоставить читателю как можно большую степень свободы действий и управления ситуацией, можно порекомендовать применение инструментов и технологий с открытым исходным кодом, чтобы настроить их в полном соответствии с вашими требованиями: что и когда вы хотите получить, в каком формате и как будут использоваться данные после их сбора. Таким образом, вы получите максимальную пользу от собранных данных.

Ниже приведен простейший пример сбора данных от IOS-устройства с использованием библиотеки языка Python `netmiko`, которая более подробно будет рассматриваться в главе 7.

```
from netmiko import ConnectHandler

device = ConnectHandler(device_type='cisco_ios', ip='csr1',username='ntc',
password='ntc123')
```

Tlgm: @it_boooks


```
output = device.send_command('show version')
print(output)
```

Основную часть содержимого переменной `output` составляет ответ команды `show version`, и у вас есть возможность выполнить синтаксический (структурный) разбор содержимого, чтобы выделить ту информацию, которая требуется в текущий момент.



В приведенном выше примере продемонстрирован способ извлечения (pulling) данных из устройств, который, возможно, не является самым оптимальным для всех сред, но остается вполне приемлемым в большинстве случаев. Следует помнить, что новейшие устройства уже поддерживают модель активной передачи данных (push model), которую часто называют потоковой телеметрией (streaming telemetry), когда само устройство в режиме реального времени непрерывно передает данные, например статистику интерфейса, на сервер приложений по вашему выбору.

Разумеется, любые извлекаемые данные могут потребовать некоторой предварительной обработки, но в конечном итоге эти трудозатраты окупаются, поскольку вы получаете именно те данные, которые необходимы, а не данные, навязанные вам каким-либо инструментом или производителем. Ведь вы читаете нашу книгу, чтобы достичь этой цели, не так ли?

В сетевых устройствах содержится огромный объем статических и непрерывно изменяющихся данных, а использование инструментальных средств с открытым исходным кодом или создание собственных инструментов предоставляет возможность доступа к этим данным. Примерами данных такого типа являются активные записи в BGP-таблице, корректировки по протоколу выбора кратчайшего пути (OSPF), обнаружение активных соседних узлов, статистические данные интерфейсов, специализированные счетчики и регуляторы состояния и даже счетчики в интегральных схемах специального назначения (ASIC), используемых на новейших платформах. Кроме того, существуют более общие факты и характеристики устройств, которые также можно собрать, например серийный номер, имя хоста, время непрерывной работы, версия ОС, аппаратная платформа и многое другое – все перечислить невозможно.



На самой начальной стадии проекта автоматизации обязательно найдите ответы на следующие вопросы: «Что лучше – создать самим, купить типовой продукт или сделать спецзаказ?» и «Что лучше – воспользоваться услугой (стать клиентом-потребителем) или непосредственно управлять рабочим процессом?»

Переходы между платформами

Переход с одной платформы на другую – это неизменно сложная задача. Переход может происходить между различными платформами одного производителя или между платформами различных производителей. Производители иногда предлагают скрипт или инструментальное средство, чтобы помочь при переходе на их платформу, но можно использовать разнообразные фор-

Tlgm: @it_boooks

мы автоматизации для создания собственных шаблонов конфигурации, как показано в приведенном выше примере, для любых типов сетевых устройств и операционных систем таким же способом, как вы могли бы сгенерировать файл конфигурации для всех производителей, взяв для этого определенный обобщенный набор входных данных (модель общих данных; common data model).

Разумеется, если производитель предлагает собственные проприетарные расширения, их также необходимо принять к рассмотрению. Хорошо, если подобное инструментальное средство поддержки перехода намного проще создать самому, чем получить от производителя. Производитель вынужден принимать во внимание все вероятные функциональные возможности устройства и обеспечить их поддержку, в то время как конкретной организации требуется лишь ограниченный набор некоторых функциональных возможностей. В действительности производители почти не уделяют внимания этому вопросу, они заняты своим аппаратным оборудованием, но не стремятся упростить его применение для сетевого оператора, управляющего средой, содержащей устройства многих производителей.

Обеспечение такого типа адаптируемости (гибкости) помогает не только при переходах между платформами, но и при восстановлении после аварий (disaster recovery, DR), поскольку широко распространенной практикой в центрах данных является наличие несколько различных взаимозаменяемых моделей для обычного режима эксплуатации и для режима DR и даже варианты с различными производителями. Если произошел аварийный сбой устройства по любой причине, а его замена влечет за собой переход на другую платформу, необходимо обеспечить возможность быстрого применения принятой модели общих данных (то есть фактически параметров как входных данных) и немедленно сгенерировать новую конфигурацию. Сейчас мы применяем термин «модель данных» (data model) в весьма свободной трактовке, но в главе 5 будет более четко определено, что такое модель данных, и более подробно описана работа с ними.

Таким образом, если выполняется переход на другую платформу, то следует продумать этот процесс на более абстрактном уровне и определить конкретные задачи, которые при этом требуется решить. Затем подумайте, что можно сделать для автоматизации этих задач, потому что только у вас, а не у крупных производителей сетевого оборудования есть стимул сделать реальностью автоматизацию, независимую от производителей. Например, рассматривайте добавление виртуальной локальной сети как абстрактный этап – о низкоуровневых командах для конкретной платформы можно подумать и потом. Здесь основная идея заключается в том, что на начальной стадии применения автоматизации чрезвычайно важно рассматривать задачи и документировать их в формате, удобном для чтения человеком, без привязок к конкретным производителям. И сделать это надо до того, как вы начнете вводить команды с клавиатуры или писать код (для целевой платформы).

Tlgm: @it_boooks

Управление конфигурацией

Как было отмечено ранее, управление конфигурацией является самым широко распространенным типом автоматизации, поэтому здесь мы ограничимся лишь кратким описанием. Следует уточнить, что под управлением конфигурацией подразумевается развертывание, передача и управление состоянием конфигурации устройства. Это относится не только к простейшим операциям, таким как подготовка к работе виртуальной локальной сети, но и к более сложным многоэтапным рабочим процессам конфигурирования стоечных (top-of-rack) коммутаторов, сетевых экранов (firewalls), балансировщиков нагрузки и развитой инфраструктуры защиты, а также к процессу развертывания трехзвенных приложений.

Как уже было сказано при предварительном ознакомлении с различными формами автоматизации, нет необходимости начинать процесс автоматизации с передачи конфигурации в устройства. Если вы тратите многие часы, передавая одно и то же изменение в определенное количество маршрутизаторов или коммутаторов, возможно, вы делаете именно то, что нужно вам.

Есть много способов приступить к автоматизации сети, но если вы начинаете с автоматизации управления конфигурацией, то помните: чем больше власть, тем больше ответственность. Еще более важно не забывать о тестировании перед развертыванием новых инструментальных средств автоматизации в реально эксплуатируемых средах.

Несколько следующих типов автоматизации, рассматриваемых в следующих разделах, представляют собой производные типы от автоматизации процесса сбора данных. Они рассматриваются отдельно, чтобы проще было понять их. Начнем с автоматизации проверок совместимости.

Совместимость

Как и многие формы автоматизации, внесение изменений в конфигурацию с помощью любого инструментального средства автоматизации связано с определенным риском. Возможно, внесение изменений вручную более рискованно, и вы могли убедиться в этом на собственном опыте или прочесть о печальном опыте коллег. При использовании автоматизации у вас есть возможность начать со сбора данных, мониторинга и формирования конфигурации – все эти операции выполняются в режиме «только для чтения» (read-only) и с минимальным риском (low-risk). Одним из практических примеров сбора данных с минимальным риском является проверка совместимости конфигураций и валидация конфигурации. Развертываемая конфигурация соответствует требованиям обеспечения безопасности? Сконфигурированы ли необходимые сетевые среды? Использование протокола XYZ запрещено? Управляя всеми развернутыми инструментальными средствами, вы получаете возможность в любой момент проверить, содержит ли какой-либо элемент значение `True` или `False`. Вполне достаточно начать с одной небольшой проверки совместимости, а затем постепенно наращивать функциональность по необходимости.

Tlgm: @it_boooks

На основе совместимости проверяемых объектов вы в конечном итоге определяете, что произойдет дальше, – возможно, действия просто фиксируются в журнале или выполняются сложные составные операции, создающие для вашего приложения возможность автоматического восстановления. Все это формы автоматизации, управляемой событиями, которой также будет уделено внимание при рассмотрении StackStorm и Salt в главе 9. В любом случае, эти формы представляют собой наилучший способ начать процесс автоматизации без лишних сложностей, но следует также помнить о потенциальных возможностях существенного расширения и усовершенствования качества автоматизации. Например, если вы всего лишь фиксируете в журнале или выводите на экран сообщения о размере максимального передаваемого блока данных (maximum transfer unit, MTU) для какого-либо интерфейса, то вы уже фактически готовы к автоматической реконфигурации с целью определения корректного значения, если текущее значение MTU не соответствует требованиям. Нужно только добавить несколько строк к уже существующим записям в журнале или к выводимым сообщениям. И повторим еще раз: начинайте с малого, но тщательно обдумывайте то, что может потребоваться в будущем.

Составление отчетов

После того как вы приступили к автоматизации сбора данных, может также потребоваться функция создания специализированных динамических отчетов. Собранная информация может стать входными данными для других задач управления конфигурацией (также управляемых событиями или в большей степени зависящих от определенных условий). Кроме того, сами по себе отчеты важны в процессе управления.

Отчеты также могут генерироваться по шаблонам, объединяемым с текущими динамически получаемыми от устройств данными, которые вставляются в шаблон, поэтому процесс создания и использования шаблонов практически аналогичен процессу создания шаблонов конфигурации, описанному выше в этой главе (напомним, что более подробно шаблоны будут рассматриваться в главе 6).

Шаблоны на основе текста просты в применении, поэтому можно генерировать отчеты в любом требуемом формате, в том числе в следующих широко распространенных форматах (список неполон):

- простые текстовые файлы;
- файлы в формате Markdown (https://daringfireball.net/projects/markdown/), которые можно просматривать на сервисе GitHub или с помощью любой программы просмотра файлов, поддерживающей формат Markdown;
- отчеты в формате HTML, развернутые на веб-сервере для быстрого доступа и просмотра.

Выбор зависит от ваших требований. Основное преимущество здесь состоит в том, что специалист, занимающийся автоматизацией сети (network automator), получает возможность создать тот тип отчета, который ему нужен в те-

Tlgm: @it_boooks

кущий момент. Вы можете использовать один набор данных для генерации различных типов отчетов, как сугубо технических, так и упрощенных для более высокого уровня управления.

В следующем разделе мы рассмотрим важность автоматизации процедур, направленных на устранение проблем и неисправностей.

Устранение проблем

Вряд ли кому-то понравится, если его постоянно вызывают для устранения возникших проблем и неисправностей, особенно за счет времени отдыха или времени, отведенного для решения других задач.

После получения прямого доступа в реальном времени к необходимым данным, когда уже не требуется ручное выполнение структурного (синтаксического) анализа этих данных, автоматизация устранения проблем и неисправностей становится реальностью.

Тщательно продумайте процесс устранения проблем и неисправностей. У вас есть личная методика? Эта методика согласована с действиями всех членов вашей группы? Выполняется ли сначала проверка уровня 2 (канальный уровень) до начала процесса устранения неисправностей на уровне 3 (сетевой уровень)? Какие конкретные шаги вы предпринимаете для устранения возникшей проблемы?

Рассмотрим пример процесса устранения проблем при работе с протоколом маршрутизации OSPF («первым выбирается кратчайший путь»):

- известны ли вам условия формирования OSPF-окружения в области сетевой среды между двумя устройствами?
- способны ли вы немедленно и точно ответить на этот вопрос и другие вопросы в 2 часа ночи или во время отпуска, проводимого на берегу моря?
- возможно, вы помните, что некоторые похожие устройства должны находиться в той же подсети, иметь то же значение MTU и согласованные таймеры, но забыли, что необходимо сделать для этих устройств, чтобы получить тот же тип OSPF-сети;
- действительно ли вы обязаны помнить все перечисленное выше, а также все соответствующие команды CLI-интерфейса для извлечения каждого фрагмента данных?

И это только небольшая часть вопросов, на которые вы должны ответить, чтобы обеспечить работу сети с протоколом OSPF.

В любой среде необходимо выполнять проверки совместимости подобного типа. Способны ли вы полностью понять функциональность запускаемого скрипта или используемого инструментального средства для валидации OSPF-окружения в сравнении с тем же процессом, выполняемым вручную? Какой вариант вы бы предпочли?

Следует еще раз подчеркнуть, что OSPF – это всего лишь «верхушка айсберга». Необходимо продумать ответы и на другие вопросы, при этом оставаясь на «верхушке»:

Tlgm: @it_boooks

- можете ли вы установить связь между конкретными сообщениями, зафиксированными в журнале, с известными условиями в сети?
- что известно о смежных соседних сегментах, доступных по протоколу BGP (протокол граничного шлюза)? Как сформированы эти соседние сегменты?
- видите ли вы все маршруты, которые, по вашему мнению, должны содержаться в таблице маршрутизации?
- что известно о конфигурации VPC и MLAG (Multi-Chassis Link Aggregation Group, группа агрегирования каналов на нескольких устройствах)?
- что известно о взаимосвязи порт–канал? Есть ли какая-либо несогласованность?
- соседние узлы (сегменты) соответствуют конфигурации порт–канал (переход на более низкий уровень – программный коммутатор vSwitch)?
- в каком состоянии находятся сетевые кабели? Все ли кабели подключены правильно?

Но, даже ответив на эту группу вопросов, вы лишь немного углубились в область задач, связанных с автоматизацией диагностики и устранения проблем и неисправностей.



На первоначальном этапе изучения всех возможных типов автоматизации начните с воображаемой системы с активной автоматической обратной связью, в которой данные собираются автоматически, затем данные автоматически обрабатываются и анализируются, после чего воспользуйтесь современными аналитическими инструментами для автоматизации процедур устранения проблем и неисправностей. Если этот начальный этап согласованно выполняется унифицированным способом, то действительно возникает активная обратная связь, полностью изменяющая способы выполнения операций, применяемые в данной организации.

Если вы являетесь «ведущим специалистом-экспертом» в группе сетевых инженеров, возможно, следует рассмотреть возможность сотрудничества с разработчиком или, по крайней мере, начать документировать свои рабочие процессы, чтобы с легкостью делиться своими знаниями и упростить предстоящее написание программного кода.

Но еще лучше начать собственный процесс автоматизации и организовать его таким образом, чтобы появилась возможность отдыхать, когда нужно и сколько нужно, передавая другим сотрудникам полномочия на устранение проблем с применением ваших методик автоматической диагностики рабочих процессов.

Теперь вы сами убедились в том, что автоматизация сети – это нечто гораздо большее, чем ускоренное развертывание конфигураций. После знакомства с некоторыми типами автоматизации мы переходим к рассмотрению разнообразных инструментов автоматизации и различных способов обмена данных приложений с сетевыми устройствами, начиная с протокола SSH, а закончим описанием NETCONF и RESTful API на основе протокола HTTP.

Tlgm: @it_boooks

Развитие уровня управления от протокола SNMP до API устройств

Если необходимо улучшить качество управления сетями и методиками выполнения повседневных рабочих операций, то усовершенствование непременно следует начать с интерфейса, управляющего устройствами более низкого уровня. В этом случае интерфейс определяет, каким образом вы и, что более важно, применяемые средства автоматизации взаимодействуют с устройствами для реализации различных типов автоматизации сети, например таких как сбор данных и управление конфигурацией.

В этом разделе приведен общий обзор разнообразных методов, доступных для организации взаимодействия с уровнем управления сетевыми устройствами, начиная с протокола SNMP и заканчивая более современными методиками и инструментами, такими как NETCONF и RESTful API. Затем будет рассматриваться развитие концепции открытых сетей, также имеющей отношение к выполнению сетевых операций и автоматизации сети.

Прикладные программные интерфейсы (API)

Настоящий сетевой инженер должен правильно воспринимать и способствовать развитию прикладных программных интерфейсов (API), а не бояться их. Следует помнить, что API – это всего лишь механизм, используемый в ПО одного устройства, для того чтобы обмениваться информацией с ПО другого устройства. В современном интернете API применяются практически повсеместно – наконец-то они стали предметом пристального внимания производителей сетевых продуктов (как аппаратных, так и программных). В самом ближайшем будущем можно ожидать, что API станут главными средствами управления сетевыми устройствами.

Более подробно конкретные сетевые API рассматриваются в главе 7, а в этом разделе приводится только обобщенный обзор нескольких различных типов API, в настоящее время широко применяемых в сетевых устройствах.

Протокол SNMP

Протокол SNMP широко применялся для управления сетевыми устройствами в течение более чем 20 лет. Вероятно, он знаком большинству читателей, поскольку и сейчас достаточно часто используется для опроса сетевых устройств с целью получения такой информации, как состояние (активно/неактивно), тип процессора, оперативной памяти, а также характеристики интерфейса.

Чтобы воспользоваться протоколом SNMP, необходим SNMP-агент на управляемом устройстве и узел/система сетевого управления (network management station, NMS), то есть устройство, работающее как сервер, который наблюдает и/или управляет подконтрольными устройствами.

Каждое управляемое сетевое устройство предоставляет набор данных, которые могут быть собраны и сконфигурированы с помощью SNMP-агента. Набор

Tlgm: @it_boooks

данных, управляемых по протоколу SNMP, описывается и моделируется через базы данных управляющей информации (management information bases, MIB). Каждая конкретная сетевая функция или характеристика может отслеживаться и управляться только в том случае, если она зафиксирована в БД MIB. Внесение изменений в конфигурацию также выполняется по протоколу SNMP. Часто не обращают внимания на то, что протокол SNMP поддерживает не только функцию `GetRequests` для мониторинга, но и функцию `SetRequests` для работы с объектами и переменными, хранящимися в БД MIB. Проблема заключается в том, что лишь немногие производители обеспечивают полную поддержку управления конфигурацией по протоколу SNMP, но даже при полной поддержке производители часто используют собственные специализированные БД MIB. Это замедляет и затрудняет процесс интеграции различных платформ управления сетями.

Как было отмечено выше, протокол SNMP существует в течение нескольких десятилетий, но создан он был не для применения в качестве программного интерфейса реального времени для сетевых устройств. Уже появились производители, предсказывающие постепенное исчезновение SNMP и замену его следующим поколением инструментальных средств управления и автоматизации. Но сейчас поддержка SNMP имеется практически в каждом сетевом устройстве, кроме того, существуют библиотеки языка Python для работы с SNMP, поэтому если необходимо собрать основную информацию с огромного количества сетевых устройств различных типов, то, возможно, имеет смысл воспользоваться протоколом SNMP.

Подобно тому, как протокол SNMP годами применялся для сетевого мониторинга, протоколы SSH/Telnet и соответствующие утилиты командной строки использовались для управления конфигурацией. Сейчас мы рассмотрим эти инструменты.

Протоколы SSH/Telnet и утилиты командной строки

Если вам когда-либо приходилось заниматься управлением сетевых устройств, то вы наверняка использовали интерфейс командной строки, чтобы вводить команды, выполняющие некоторые операции с устройством. Возможно, вы вводили команды в консоли или в сеансах Telnet и SSH. В главе 1 уже отмечалось, что практический переход от Telnet к SSH стал, вероятно, самым большим достижением в сфере выполнения сетевых операций за последнее десятилетие, но это достижение не коснулось самих операций. Этот переход обеспечил защиту процесса обмена информацией между сетевыми устройствами с помощью шифрования.

Важнее всего понять, что организация управления устройствами через интерфейс командной строки была выбрана потому, что интерфейс командной строки был создан для людей. Поддержка CLI обеспечивалась в устройствах для повышения удобства выполнения операций человеком. Интерфейс командной строки не был предназначен для обмена данными непосредственно между устройствами (то есть не предназначался для написания сетевых скриптов и для автоматизации).

Tlgm: @it_boooks

Если выполнить команду `show` в интерфейсе командной строки какого-либо устройства, то в ответ вы получите необработанный текст, который совершенно не структурирован. Для проведения синтаксического разбора такого текста наилучшим решением является использование программного канала (pipe, |) и ключевых слов, таких как `grep`, `include` и `begin`, для поиска требуемых строк в конфигурации. В качестве примера можно привести команду проверки описания сетевого интерфейса `show interface Eth1 | include description`. А если потребуется узнать количество ошибок CRC на этом интерфейсе после выполнения команды `show interface` в скрипте, то придется воспользоваться регулярными выражениями или анализировать полученные данные вручную. Такой подход неприемлем.

Но если в нашем распоряжении имеется только интерфейс командной строки, то его нужно использовать. Именно поэтому за последние два десятилетия появилось огромное количество платформ сетевого управления и специализированных скриптов, которые выполняют операции управления и автоматизации, используя интерфейс командной строки в сеансах SSH с применением expect-скриптов и ручного синтаксического разбора. Это не означает, что комбинация SSH/CLI делает автоматизацию невозможной, просто автоматизация с использованием этих инструментов становится чрезвычайно утомительной, а вероятность совершения ошибок резко возрастает.

Производители начали понимать создавшуюся ситуацию, и в настоящее время большинство новейших аппаратных сетевых платформ обеспечено определенным типом API, который упрощает обмен данными между устройствами (многие API пока не доработаны полностью, поэтому рекомендуется тщательное тестирование API выбранного устройства). Тем самым существенно упрощается подход к автоматизации, который к тому же в большей степени согласуется с основными принципами разработки программного обеспечения.

После краткого обзора широко известных протоколов, таких как SSH и SNMP, переходим к рассмотрению NETCONF – API, который становится все более распространенным средством для решения задач автоматизации сети.

NETCONF

NETCONF – это многоуровневый протокол управления сетевыми устройствами. На самом высоком уровне его можно сравнить с протоколом SNMP, так как оба протокола используются для внесения изменений в конфигурацию и для извлечения данных из сетевых устройств.

Разумеется, различия проявляются в деталях. Некоторые особенности высокого уровня перечислены ниже, но более подробно NETCONF рассматривается в главе 7. Краткая характеристика протокола NETCONF:

- это сетевой протокол передачи данных с установлением соединения, который в качестве транспортного средства обычно использует SSH;
- данные, передаваемые между клиентом NETCONF (инструмент/скрипт автоматизации) и сервером NETCONF (сетевое устройство), закодирова-

Tlgm: @it_boooks

ны в формате XML. Если вы незнакомы с форматом XML, не беспокойтесь, этот формат будет описан в главе 5;

- вызовы удаленных процедур (remote procedure calls, RPC) закодированы в XML-документе, передаваемом в устройство, которое обрабатывает эти вызовы. Элемент `<rpc>` используется для обозначения запроса NETCONF, передаваемого от клиента серверу. В таком контексте следует трактовать вызовы удаленных процедур как заранее подготовленные операции, выполняемые на сетевом устройстве. RPC – это способ, которым клиент сообщает серверу, какую структуру и какой тип запроса необходимо обработать и выполнить;
- допустимые вызовы удаленных процедур (RPC) напрямую отображаются в соответствующие поддерживаемые NETCONF операции и функциональные возможности для конкретных устройств. Например, при необходимости внесения изменений в конфигурацию устройства используется операция `edit-config`. Для извлечения конфигурационных данных применяется операция `get` или `get-config`. Эти операции в XML-документе «завернуты» в теги элемента `<rpc>` и в таком виде передаются в устройство.

Кроме того, NETCONF предлагает значение (параметр), обеспечивающее поддержку изменений, основанных на транзакциях. Если вносится более одного изменения в одном сеансе NETCONF или в одном XML-документе и одно из этого набора изменений завершается сбоем, то весь набор изменений не применяется к данному устройству (разумеется, обычно можно поменять и эти типы настроек). Такой подход отличается от последовательной передачи команд из интерфейса командной строки, при которой можно в итоге получить незавершенную конфигурацию из-за опечатки или неправильной команды.

Это краткое введение, а более подробно NETCONF будет рассматриваться в главе 7.



Следует отметить, что поддержка NETCONF (или любой другой методики передачи данных) двумя различными аппаратными платформами не означает, что они совместимы с точки зрения разработчиков ПО и инструментальных средств. Даже если предположить, что обе аппаратные платформы поддерживают одни и те же функциональные возможности и характеристики NETCONF, каждый производитель чаще всего использует собственный особый способ моделирования данных. Моделирование данных – это способ и форма представления в устройстве данных о его состоянии и о конфигурации. В главе 5 подробно рассматривается представление данных в форматах JSON и XML, а также использование широко распространенного языка моделирования данных YANG.

Прикладные программные интерфейсы RESTful API

Аббревиатура REST означает REpresentational State Transfer (передача состояния представления) и представляет собой стиль проектирования и разработки сетевой архитектуры программного обеспечения. Таким образом, системы, реализующие этот стиль, следующие его требованиям и не нарушающие ограничений основанной на REST архитектуры, обозначаются специальным термином RESTful.

Tlgm: @it_boooks

В сетевой среде устройствами, которые чаще всего предоставляют соответствующий API и следуют архитектурному стилю REST, являются сетевые контроллеры. Следует отметить, что существуют сетевые устройства, поддерживающие как RESTful API, так и обобщенный API на основе протокола HTTP.

Сами термины REST и RESTful API являются относительно новыми терминами в сетевой отрасли, но вы и ранее уже взаимодействовали со многими RESTful-системами буквально каждый день, когда путешествовали по интернету с помощью веб-браузера. В начале раздела было указано, что REST – это стиль разработки сетевой архитектуры программного обеспечения. Этот стиль основан на модели клиент–сервер без сохранения состояния (stateless client-server model), когда клиент сам контролирует свой сеанс, но состояние клиента и текущий контекст не сохраняются на сервере. И чаще всего в качестве транспортного протокола нижележащего уровня используется HTTP. Именно с такими системами вы постоянно встречались в интернете.

Это означает, что RESTful API функционирует почти так же, как системы на основе протокола HTTP. Во-первых, необходим веб-сервер, доступный по своему URL (то есть SDN-контроллер или сетевое устройство для обмена данными с ним), во-вторых, необходимо обеспечить отправку соответствующего HTTP-запроса на этот URL.

Например, если нужно извлечь список устройств из SDN-контроллера, то достаточно отправить запрос HTTP GET на URL этого контроллера, который может выглядеть приблизительно так: `http://1.1.1.1/v1/devices`. Ответом на такой запрос должны быть структурированные данные некоторого типа, например XML или JSON (см. главу 5).

Здесь мы не рассматриваем некоторые другие вопросы, такие как аутентификация, шифрование данных и способы отправки HTTP-запроса при внесении изменений в конфигурацию (HTTP PUT/POST/PATCH). В этом разделе дается лишь краткое представление стиля REST и RESTful API, а более подробное описание вы найдете в главе 7.

В следующем разделе мы кратко рассмотрим влияние концепции открытых сетей на управление сетевыми устройствами в целом.

Влияние концепции открытых сетей

В настоящее время наблюдается стремление сделать открытым практически все – открытый исходный код, открытые сети, Open API, OpenFlow, Open Compute, Open vSwitch, OpenDaylight, OpenConfig и т. д. Точное определение термина «открытый» (open) можно обсуждать бесконечно, но об одном можно сказать вполне определенно: концепция открытых сетей – это реальный путь ко всем возможным усовершенствованиям сетевых операций и автоматизации сети.

Благодаря развитию этой концепции сейчас мы можем наблюдать весьма существенные изменения в сетевых устройствах, и это стало основной причиной для написания данной книги.

Tlgm: @it_boooks

Во-первых, многие устройства обеспечивают непосредственную поддержку языка программирования Python (прямо «из коробки»). Это означает, что можно воспользоваться динамическим интерпретатором Python (Python Dynamic Interpreter) и выполнять скрипты на языке Python локально на каждом сетевом устройстве. Вы по достоинству оцените это преимущество, когда будете более подробно знакомиться с языком Python в главе 4.

Во-вторых, сейчас многие устройства поддерживают более надежный и стабильный API, отличающийся от протоколов SNMP и SSH. Например, в предыдущем разделе рассматривались NETCONF и RESTful API на основе протокола HTTP. Один или оба этих API поддерживаются многими новейшими специализированными операционными системами для устройств, созданными за последние 18–24 месяца. Напомним еще раз, что более подробно API устройств рассматриваются в главе 7.

Наконец, в сетевых устройствах в настоящее время используется больше внутренних механизмов ОС Linux, которые ранее были скрыты от сетевых операторов. Сейчас вы можете воспользоваться командной оболочкой bash прямо на сетевых устройствах и выполнять команды, такие как `ifconfig`, писать bash-скрипты и устанавливать инструментальные средства мониторинга и конфигурирования с помощью менеджеров пакетов, таких как `apt` и `yum`. Все это подробно описано в главе 3.

Хотя концепция открытых сетей не всегда означает обеспечение взаимодействия, очевидно, что сетевые устройства и контроллеры, сами по себе открытые в большей степени для программного взаимодействия, лучше подходят для массовой сетевой автоматизации. Сейчас появилось множество API сетевых устройств, которых не было еще несколько лет назад, – от NX-API (Cisco), eAPI (Arista) и IOS-XE RESTCONF/NETCONF (Cisco) до любых новейших SDN-контроллеров, обладающих собственным API. В конечном итоге сетевые операторы получили возможность в полной мере управлять сетями и уменьшать количество ныне имеющихся функциональных недостатков с помощью этих современных API.

Автоматизация сети в эпоху SDN

Теперь рассмотрим непреходящую важность автоматизации сети даже при наличии реально применяемых решений для контроллеров, таких как OpenDaylight или коммерческих предложений, например Cisco ACI или VMware NSX. Конкретные операции, выполняемые контроллерами в сети, такие как функционирование в качестве панели управления или управление стратегией и конфигурацией, не относятся к теме этого раздела.

Такие контроллеры становятся неотъемлемым компонентом сетевых архитектур следующего поколения. Производители – Cisco, Juniper, VMware, Big Switch, Plexxi, Nuage, Viptela и многие другие – предлагают платформы контроллеров для своих решений следующего поколения, не говоря уже о контроллерах с открытым исходным кодом, таких как OpenDaylight и OpenContrail.

Tlgm: @it_boooks

Практически каждый контроллер, представленный на рынке, предъявляет вполне надежный RESTful API, поэтому автоматизация таких контроллеров весьма проста. Несмотря на то что сами контроллеры по своей сущности упрощают управление и наблюдение с помощью единственной визуальной панели, вы можете продолжать применение практики внесения изменений вручную через GUI контроллера с большой вероятностью совершения ошибок. Если развернуто несколько комплектов контроллеров от одного или от различных производителей, то проблемы при внесении изменений вручную, при устранении проблем и неисправностей, при сборе данных остаются нерешенными.

Завершая главу, важно отметить, что даже в новейшую эпоху SDN-архитектур и сетевых решений на основе контроллеров сохраняется необходимость в автоматизации, в усовершенствовании операций и в обеспечении предсказуемости получаемых результатов.

Резюме

В этой главе представлен общий обзор сетевой автоматизации и ее различных типов, дана краткая вводная информация о широко распространенных API сетевых устройств, включая SNMP, CLI/SSH и более важные в настоящее время NETCONF и RESTful, а также сказано несколько слов о языке сетевого моделирования YANG, который более подробно будет описан в главе 5.

Глава завершается кратким рассмотрением влияния концепции открытых сетей на автоматизацию сети и выполнение сетевых операций. В заключительном разделе отмечено важное значение автоматизации сети даже при массовом внедрении и развертывании SDN-контроллеров.

В каждой последующей главе мы будем постепенно углублять и расширять знания о каждой технологии, приводя полезные практические примеры везде, где это возможно, но в то же время будет подчеркиваться значимость человеческого фактора, организации рабочего процесса и культуры, требуемой для правильного внедрения полноценных эксплуатационных сред и конвейеров процессов автоматизации. Глава 11 полностью посвящена человеческому фактору и профессиональной культуре.

Tlgm: @it_boooks

Глава 3

Операционная система Linux

Главная цель этой главы – помочь читателям познакомиться с основами Linux, операционной системы, которая становится все более распространенной в среде специалистов, занимающихся сетями. Возможно, у вас возник вопрос: почему в эту книгу включена глава с описанием ОС Linux? По каким причинам Unix-подобная операционная система Linux должна применяться для обеспечения сетевой программируемости и автоматизации сетей?

Изучение ОС Linux с точки зрения автоматизации сети

Если взглянуть на ОС Linux с точки зрения автоматизации сети, то можно обнаружить несколько причин, по которым мы считаем эту тему важной.

Во-первых, некоторые современные сетевые операционные системы (NOS) основаны на ОС Linux, хотя иногда в них используется специализированный интерфейс командной строки (CLI), поэтому они внешне отличаются от Linux. Другие, напротив, явно используют внутренние механизмы и инструменты Linux, в частности командные оболочки, такие как bash.

Во-вторых, некоторые новые компании и организации продвигают на рынок полноценные дистрибутивы Linux, специально ориентированные на работу с сетевым оборудованием. Например, недавно группа OpenCompute Project (OCP) выбрала Open Network Linux (ONL) в качестве основы для создания сетевых ОС, обладающих всеми преимуществами Linux (Switch Light компании Big Switch – пример Linux-подобной сетевой ОС на основе ONL). Другой пример: компания Cumulus Networks предлагает Cumulus Linux на основе дистрибутива Debian в качестве сетевой ОС для поддержки аппаратных платформ. Для современного сетевого инженера становится неизбежной необходимость изучения ОС Linux для создания конфигурации своей сети.

Третья и последняя причина – многие инструментальные средства, описываемые в данной книге, ведут свое происхождение из ОС Linux или требуют для своей работы именно эту операционную систему. Например, для Ansible

Tlgm: @it_boooks

(см. главу 9) требуется установленная программная среда языка Python (см. главу 4). По нескольким причинам, которые мы объясним в главе 9, при автоматизации сетевого оборудования с помощью Ansible это инструментальное средство обычно запускается из подключенной к сети системы под управлением ОС Linux, а не на самом сетевом оборудовании. При использовании языка Python для сбора и обработки данных из сетевого оборудования чаще всего программная среда Python запускается на системе под управлением ОС Linux.

По указанным выше причинам мы считаем важным и обоснованным включение в книгу главы с описанием ОС Linux в следующих целях:

- краткое описание истории создания ОС Linux;
- краткое описание концепции дистрибутивов Linux;
- знакомство с bash, наиболее распространенной командной оболочкой ОС Linux;
- изучение основ работы с сетями в ОС Linux;
- более подробное рассмотрение некоторых расширенных сетевых функциональных возможностей ОС Linux.

Следует подчеркнуть, что эта глава не является полноценным учебником по ОС Linux или по командной оболочке bash. Она представляет собой «отправной пункт» для изучения ОС Linux в контексте сетевой программируемости и автоматизации сети. А начнем мы с краткого обзора источников и истории создания ОС Linux.

Краткая история создания ОС Linux

В истории создания операционной системы Linux необходимо выделить два основных направления.

Первое направление сформировалось в начале 1980-х гг., когда Ричард Столлман (Richard Stallman) организовал GNU Project для создания свободной (общедоступной и бесплатной) Unix-подобной операционной системы. GNU означает «GNU's Not Unix» – этот рекурсивный акроним Столлман придумал для описания сущности свободной Unix-подобной ОС, которую он пытался создать. Усилиями участников GNU Project и самого Столлмана была сформулирована лицензия GNU General Public License (GPL). GNU Project начал успешно создавать многочисленные свободные версии обширного набора утилит и приложений Unix, но ядро, известное как GNU Hurd и специально предназначенное для новой операционной системы GNU Project, так и не достигло состояния полной готовности.

Второе направление связано с именем Линуса Торвальдса (Linus Torvalds), который написал клон операционной системы Minix в 1991 году как первоначальный прототип будущей ОС Linux. Отсутствие свободно распространяемого ядра операционной системы на тот момент привело к тому, что начатая Торвальдсом работа сразу получила быстро развивающуюся поддержку, и уже в 1992 году версия 0.99 была выпущена под лицензией GNU GPL. С этого момента ядро, написанное Торвальдсом (известное под именем Linux), становит-

Tlgm: @it_boooks

ся принятым по умолчанию ядром операционной системы с комплектом программного обеспечения, созданным организацией GNU Project.

Поскольку Linux изначально представлял собой только ядро ОС и для формирования полноценной операционной системы требовался комплект ПО GNU Project, некоторые люди решили, что весь комплект ПО в совокупности должен называться GNU/Linux. Некоторые организации и сейчас придерживаются этого мнения (например, Debian). Тем не менее большинство людей называет всю операционную систему в целом просто Linux, и в этой книге в дальнейшем также будет использоваться название Linux.

Дистрибутивы Linux

В предыдущем разделе было отмечено, что операционная система Linux состоит из ядра и обширного набора инструментальных средств с открытым исходным кодом, разработанного преимущественно в рамках GNU Project. Объединение ядра с комплектом программного обеспечения с открытым исходным кодом привело к созданию дистрибутивов (distributions, часто просто distros) Linux. Дистрибутив представляет собой совокупность ядра Linux (иногда нескольких версий ядер) и специально подобранного комплекта утилит, приложений и программных пакетов с открытым исходным кодом, которые взаимосвязаны и распространяются (отсюда и название дистрибутив – от distribution – распространение) вместе, как единый программный комплекс. В процессе развития ОС Linux появлялись и приобретали широкую известность многие дистрибутивы (пример: некогда популярный, но сейчас несколько подзабытый Slackware), но на сегодняшний день существуют две основные ветви дистрибутивов Linux: Red Hat/CentOS и Debian, а также производные от него дистрибутивы.

Red Hat Enterprise Linux, Fedora и CentOS

Компания Red Hat была одним из первых распространителей ОС Linux, а сейчас стала одной из самых влиятельных и коммерчески успешных организаций на рынке Linux, поэтому вполне естественно, что одной из основных ветвей дистрибутивов Linux стал именно Red Hat и основанные на нем производные дистрибутивы.

Компания Red Hat предлагает коммерческий дистрибутив под названием Red Hat Enterprise Linux (RHEL) с дополнительными контрактами на техническую поддержку RHEL. Сейчас многие организации используют RHEL, потому что автором этого дистрибутива является компания Red Hat, которая главное внимание уделяет стабильности и надежности, предлагает варианты комплексной технической поддержки, а кроме того, активно поддерживается другими производителями программного обеспечения.

Но быстрый темп развития ОС Linux и сообщества разработчиков и пользователей программного обеспечения с открытым исходным кодом для Linux за-

Tlgm: @it_boooks

частую не соответствует более медленному и более методичному темпу развития, необходимому для обеспечения стабильности и надежности дистрибутива RHEL. Для устранения этого противоречия компания Red Hat создала дистрибутив Fedora, который называют «upstream-дистрибутивом», потому что разработки RHEL и основанных на нем дистрибутивов включаются в Fedora, затем «спускаются» обратно в эти и другие программные продукты. В сотрудничестве с более широким сообществом разработчиков ПО с открытым исходным кодом проект Fedora испытывает и тестирует новые версии ядра, новые функциональные возможности ядра, новые инструментальные средства управления пакетами и прочие нововведения. Все эти новые возможности и свойства тестируются и оцениваются в дистрибутиве Fedora перед последующим внедрением их в ориентированный на производственную эксплуатацию дистрибутив RHEL. Поэтому можно сказать, что Fedora используется разработчиками и прочими специалистами и пользователями, которым необходимо «все самое свежее и самое лучшее», но крайне редко можно обнаружить использование Fedora на предприятиях и в организациях для реальной работы.

Несмотря на то что RHEL и его варианты можно получить только от компании Red Hat на коммерческой основе, лицензия GNU GPL, под которой разрабатывается и распространяется ОС Linux, требует в обязательном порядке, чтобы исходные коды дистрибутивов Red Hat были открыты и доступны всем желающим. Группа энтузиастов, которым нужна была стабильность и надежность RHEL, но без коммерческой выгоды, извлекаемой Red Hat, взяла за основу исходные коды дистрибутива RHEL и создала свой дистрибутив CentOS. (Название CentOS представляет собой сокращение фразы Community Enterprise OS.) CentOS распространяется свободно, без какой-либо оплаты, но, как и многие программные пакеты с открытым исходным кодом, не обеспечивает техническую поддержку в какой-либо форме. Для многих организаций в большинстве случаев вполне достаточно поддержки, предоставляемой сообществом open source, поэтому довольно часто можно наблюдать использование CentOS в различных рабочих средах, в том числе и в корпоративных.

Все перечисленные выше дистрибутивы (RHEL, Fedora, CentOS) объединяет общий формат пакетов (package format). Когда дистрибутивы Linux начинали формироваться, главной задачей являлся выбор способа компоновки пакетов программ и стыковка этих пакетов с ядром Linux. Из-за огромного объема свободного программного обеспечения для ОС Linux стала очевидна неэффективность включения всех доступных программ в дистрибутив, к тому же пользователям не нужна была установка абсолютно всех программных компонентов. Но если устанавливается не все программное обеспечение, то каким образом Linux-сообщество должно было решать проблему зависимостей? Зависимость (dependency) – это компонент программного обеспечения, требуемый для обеспечения работы другого компонента программного обеспечения в компьютерной системе. Например, некоторая программа написана на языке Python, поэтому для нее, разумеется, необходима установка программной

Tlgm: @it_boooks

среды Python. Но для установки программной среды Python может потребоваться установка других компонентов программного обеспечения и т. д. Как один из первых создателей дистрибутивов компания Red Hat разработала способ объединения файлов, необходимых для работы программного компонента, с включением дополнительной информации о зависимостях от других программных компонентов в одном пакете, то есть был создан формат пакета (package format). Этот формат пакетов известен под названием RPM, возможно, связанным с названием инструментального средства, изначально используемого для работы с пакетами: RPM Manager (ранее Red Hat Package Manager), а соответствующий выполняемый файл назывался просто `rpm`. Все упомянутые выше дистрибутивы – RHEL, CentOS, Fedora – используют формат пакетов RPM по умолчанию, хотя инструмент для работы с этими пакетами со временем изменялся и усовершенствовался.



Преемники менеджера пакетов RPM

Мы отметили, что RPM изначально был представлен как настоящий менеджер пакетов, работавший с rpm-пакетами. Впоследствии большинство дистрибутивов на основе формата RPM заменило утилиту rpm на более новые менеджеры пакетов, которые лучше разбирались с зависимостями, разрешали конфликты и позволяли устанавливать и удалять программные компоненты в работающей Linux-системе. Например, RHEL/CentOS/Fedora сначала перешли на использование инструментального средства yum (Yellowdog Updater, Modified), а сейчас снова переходят на новый инструмент – dnf (Dandified YUM).

Формат пакетов RPM используют и другие дистрибутивы, например Oracle Linux, Scientific Linux и различные варианты и производные SUSE Linux.



Переносимость формата пакетов RPM

Поскольку многие дистрибутивы Linux используют один и тот же формат пакетов RPM, можно было бы подумать, что RPM-пакеты являются переносимыми между этими дистрибутивами Linux. Теоретически это возможно, но на практике полная переносимость встречается редко. Обычно этому препятствуют небольшие расхождения в вариантах имен пакетов и в версиях пакетов для разных дистрибутивов, из-за которых обработка зависимостей и разрешение конфликтов становятся практически невозможными.

Debian, Ubuntu и другие производные дистрибутивы

Debian GNU/Linux – это дистрибутив, создаваемый и сопровождаемый организацией Debian Project. О создании организации Debian Project ее основатель Иэн Мердок (Ian Murdock) официально объявил 16 августа 1993 года, а создание дистрибутива Debian GNU/Linux спонсировалось организацией Free Software Foundation через GNU Project с ноября 1994 года по ноябрь 1995 года. На сегодняшний день Debian остается единственным крупным дистрибьютором ОС Linux, который не извлекает коммерческой выгоды из своей деятельности. Выпуски Debian GNU/Linux с версии 1.1 используют в качестве кодовых названий имена персонажей мультипликационного сериала «История игрушек» (Toy Story). Debian GNU/Linux 1.1, вышедший в июне 1996 года, получил кодо-

Tlgm: @it_boooks

вое имя Buzz. Самая последняя стабильная версия Debian GNU/Linux, опубликованная в июне 2017 года, имеет номер 9.0 и кодовое имя Stretch.

Дистрибутив Debian GNU/Linux предлагает три ветви: стабильную (stable), тестовую (testing) и нестабильную (unstable). Тестовая и нестабильная ветви являются непрерывно обновляемыми (rolling release) и в конечном итоге становятся следующей стабильной ветвью. Такая методика обычно обеспечивает весьма высокое качество окончательной стабильной версии и, вероятно, является одной из причин, по которым многие другие дистрибутивы берут за основу Debian GNU/Linux (то есть являются производными от него).

Одним из самых известных призводных дистрибутивов на основе Debian является Ubuntu Linux, появившийся в апреле 2004 года как результат работы, бóльшая часть которой была проделана Canonical Ltd., компанией, основанной Марком Шаттлвортом (Mark Shuttleworth). В октябре 2004 года вышла первая официальная версия Ubuntu под номером 4.10 (4 обозначает год, а 10 – номер месяца) с кодовым именем Warty Warthog. Все кодовые имена версий Ubuntu составлены из прилагательного и названия животного, причем начальные буквы обоих слов одинаковы (Worty Warthog, Hoary Hedgehog, Breezy Badger и т. д.). С самого начала дистрибутив Ubuntu был позиционирован как удобный для пользователя «настольный дружественный Linux», но сейчас предлагается не только настольная версия, но и серверная и мобильная версии. Выпуски версий Ubuntu фиксированы по времени – новые версии публикуются каждые шесть месяцев, и обеспечивается долговременная поддержка (long-term support, LTS) версий в течение двух лет. LTS-релизы поддерживаются компанией Canonical и сообществом Ubuntu в общей сложности в течение пяти лет со дня выпуска. Все версии Ubuntu основаны на пакетах, взятых из нестабильной ветви Debian, поэтому Ubuntu классифицируется как дистрибутив, производный от Debian.

Несколько слов о пакетах: как и дистрибутивы на основе RPM, Debian и его производные (возможно, правильнее всего использовать выражение «производные от Debian» для их характеристики) объединяют общий формат пакетов – Debian package format (обозначаемый как расширение файлов *.deb*). Основатели Debian Project создали формат пакетов DEB и инструментальное средство `dpkg` для решения тех же задач, которые Red Hat пытался решить с помощью формата пакетов RPM. Дистрибутивы на основе Debian также усовершенствовали свои инструменты управления пакетами: после практического применения `dpkg` сначала появилось инструментальное средство `dselect`, затем авторы дистрибутива перешли на использование `apt` (и таких вспомогательных программ, как `apt-get` и `aptitude`).



Переносимость пакетов Debian

Как и в случае с пакетами RPM, факт использования формата пакетов Debian многочисленными дистрибутивами не означает, что эти пакеты всегда переносимы между дистрибутивами. Небольшие изменения в именах пакетов, версиях, в путевых именах файлов и в некоторых других деталях, как правило, чрезвычайно затрудняют перенос пакетов между дистрибутивами, зачастую делая его вообще невозможным.

Tlgm: @it_boooks

Главной функциональной характеристикой инструментальных средств на основе apt является возможность извлечения пакетов из одного или нескольких удаленных репозиториев (repositories), представляющих собой онлайновые хранилища пакетов Debian[1]. Кроме того, инструменты на основе apt обеспечивают улучшенное определение зависимостей, разрешение конфликтов и методику установки и удаления пакетов.

Другие дистрибутивы Linux

Существует множество других общедоступных дистрибутивов, но описанные выше две ветви – Red Hat/Fedora/CentOS и Debian/Ubuntu – в настоящее время представляют подавляющее большинство установленных в организациях дистрибутивов Linux. Поэтому в текущей главе мы сосредоточимся на этих двух ветвях. Если вы используете другой дистрибутив, не принадлежащий к одной из этих ветвей, например работаете с дистрибутивом SUSE Enterprise Linux, то необходимо помнить о том, что возможны небольшие различия между информацией, содержащейся в данной книге, и информацией, предназначенной для вашего конкретного дистрибутива. Для уточнения подробностей следует обратиться к документации используемого дистрибутива.

После краткого обзора истории Linux и основных дистрибутивов переходим к подробному рассмотрению работы с ОС Linux, причем главное внимание будет уделено взаимодействию с операционной системой через командную оболочку.

Работа в ОС Linux

Как весьма распространенную серверную операционную систему, Linux можно использовать разнообразными способами в сетевой среде. Например, можно распределять IP-адреса через сервер DHCP на основе Linux, получать доступ к управляемому Linux веб-серверу, на котором работает HTTP-сервер Apache или Nginx, или использовать сервер DNS (Domain Name Server) под управлением Linux для преобразования доменных имен в IP-адреса. Разумеется, можно привести гораздо больше примеров, здесь перечислено лишь несколько вариантов. Но с учетом тематики книги наше внимание сосредоточено главным образом на взаимодействии с ОС Linux через командную оболочку (shell).

Командная оболочка предоставляет интерфейс командной строки, с помощью которого многие пользователи взаимодействуют с Linux-системой. Для ОС Linux предлагается несколько командных оболочек, но наиболее распространенной и чаще всего используемой является bash, Bourne Again Shell (ее имя происходит от одной из ранних командных оболочек Unix – Bourne Shell). В подавляющем большинстве случаев пользователи работают с ОС Linux

[1] Средства управления пакетами RPM (и пакетами большинства других дистрибутивов) также предоставляют такую возможность. Странно, что авторы особо выделяют эту характеристику именно в Debian. – *Прим. перев.*

Tlgm: @it_boooks

через bash, если только система не сконфигурирована специально для использования другой командной оболочки. В этом разделе представлено достаточно информации, чтобы начать работу в консоли Linux-системы, при этом предполагается, что вы используете bash в качестве командной оболочки. При работе с другой командной оболочкой следует помнить о возможных небольших различиях в выполнении команд и в получаемых результатах.



Подробное справочное руководство по bash

Командная оболочка bash – это тема отдельной книги. И такая книга действительно существует, уже в третьем издании. Если вы хотите узнать о командной оболочке bash больше, чем предлагает наша книга, то рекомендуем изучить третье издание Learning the bash Shell издательства O'Reilly.

Информация о работе с ОС Linux разделена на четыре основные части:

- перемещение по файловой системе;
- работа с файлами и каталогами;
- работа с программами;
- работа с фоновыми сервисами, то есть с демонами (daemons).



Здесь излагаются основы на начальном уровне обучения

Этот раздел в основном ориентирован на пользователей, которые являются новичками в ОС Linux (многие сетевые инженеры и IT-профессионалы в большей степени знакомы с Microsoft Windows). Если вы хорошо знакомы с ОС Linux, то можете пропустить данный раздел.

Начнем с изучения средств перемещения по файловой системе.

Перемещение по файловой системе

В ОС Linux используется файловая система с единственным корнем (single-root filesystem), то есть все устройства хранения данных, все каталоги и файлы в системе принадлежат одному пространству имен, обозначаемому просто как / («слеш»). (Когда вы видите слеш /, мысленно называйте его root (корень).) Это полная противоположность системам, подобным Microsoft Windows, в которых каждому устройству хранения данных (дисковым накопителям и т. п.) соответствует собственный отдельный корень (обозначаемый буквами латинского алфавита, например, C:\ или D:\). Отметим, что в ОС Windows существует возможность монтирования диска (устройства хранения данных) на папку (folder), но такая практика не является общепринятой.



Все интерпретируется как файл

ОС Linux следует концепции Unix – все интерпретируется как файл. Это касается и устройств хранения данных (интерпретируемых как блоковые устройства (block devices)), и портов компьютера (например, последовательных портов – как байт-ориентированных устройств (byte devices)), и даже устройств ввода/вывода. Таким образом, важность файловой системы с единственным корнем, в которой на равных содержатся устройства и объекты хранения данных, становится еще более значимой.

Tlgm: @it_boooks

Как и большинство других операционных систем, Linux использует концепцию каталогов (directories) (в некоторых ОС их называют папками (folders)) для группирования файлов в файловой системе. Каждый файл размещается в каталоге, следовательно, для каждого файла существует единственный в своем роде путь (path) к его месту расположения. Для обозначения пути к файлу начинают с корня (с корневого каталога /) и перечисляют все каталоги, в которые надо войти, чтобы добраться до этого файла. Каталоги отделяются друг от друга символом обычного слеша (/). Например, утилита командной строки `ping` чаще всего размещается в каталоге *bin*, расположенном в корневом каталоге. Таким образом, путь к выполняемому файлу `ping` обозначается следующим образом: */bin/ping*.

Другими словами, путь начинается в корневом каталоге (/), продолжается переходом в каталог *bin/*, в котором обнаруживается файл с именем *ping*. Аналогичным образом в Debian Linux версии 8.1 утилита `arp` для работы с объектами типа ARP (Address Resolution Protocol) расположена (то есть имеет полное путевое имя) в локации */usr/sbin/arp*.

Описанная выше концепция пути (путевого имени) становится важной, когда мы начинаем изучать возможности, предоставляемые командной оболочкой bash для перемещения по файловой системе. Промпт (prompt), или текст, который bash выводит, чтобы сообщить о готовности принять ввод пользовательской команды, сообщает, в каком месте файловой системы вы находитесь в текущий момент[1]. В системе Debian 8.1 по умолчанию промпт выглядит так:

```
vagrant@jessie:~$
```

Пользователей, не имеющих опыта работы в Linux, может смутить символ тильда (~), следующий за выражением `vagrant@jessie:` в этом примере промпта. В командной оболочке bash символ тильда является сокращенным обозначением домашнего каталога (home directory) пользователя. У каждого пользователя есть собственный домашний каталог – место для хранения личных файлов, программ и прочего контента, предназначенного только для этого пользователя. Для упрощенной ссылки на любой домашний каталог bash использует символ тильда как сокращенное обозначение. Вернемся к приведенному выше примеру промпта и более подробно рассмотрим его отдельные составные части.

1. Первая часть промпта перед символом `@` – это имя текущего пользователя (в данном случае `vagrant`).
2. Вторая часть промпта, сразу после символа `@` – это текущее имя хоста, определенное в системе, в которой вы работаете (в данном случае `jessie`).
3. После символа двоеточия указан текущий каталог, в нашем случае это ~, то есть пользователь `vagrant` в настоящий момент находится в своем домашнем каталоге.

---

[1] Это зависит от начальных настроек bash. Достаточно часто промпт представляет собой только символ доллара ($). Пользователь может сам настроить вид промпта. – *Прим. перев.*

Tlgm: @it_boooks

4. Даже завершающий символ доллара $ имеет особое значение – он сообщает, что текущий пользователь (vagrant) не обладает правами суперпользователя (root). Для пользователя с правами суперпользователя символ $ заменяется на символ # (хеш-символ, или октоторп; в русскоязычной среде его часто называют диез, или просто решетка). Такой подход аналогичен способу обозначения промпта для сетевого устройства, например маршрутизатора или коммутатора, когда промпт может изменяться в зависимости от уровня привилегий текущего пользователя.



Используемая рабочая среда

На протяжении всей этой главы вы увидите разнообразные промпты Linux, похожие на приведенный в рассмотренном выше примере. Мы используем инструментальное средство Vagrant (http://www.vagrant.com) для упрощения создания разнообразных рабочих сред Linux – в нашем случае это Debian GNU/Linux 8.1 (кодовое имя Jessie), Ubuntu Linux 14.04 LTS (кодовое имя Trusty Tahr) и CentOS 7.1.

В системе CentOS 7.1 промпт по умолчанию выглядит следующим образом:

```
[vagrant@centos ~]$
```

Это очень похоже на промпт из предыдущего примера, и здесь сообщается та же информация, но в несколько другом формате. Здесь тоже указаны текущий пользователь (vagrant), имя хоста (centos), текущий каталог (~) и действующие права доступа текущего активного (зарегистрированного в системе) пользователя ($).

Использование символа тильда удобно для сохранения краткости промпта, когда пользователь находится в своем домашнем каталоге, но что, если вы не знаете полное путевое имя своего домашнего каталога? То есть вам неизвестно, где именно в файловой системе расположен ваш домашний каталог. В подобных ситуациях, когда необходимо определить полное путевое имя текущей локации, предлагается выполнить команду pwd (print working directory), которая выводит требуемую информацию следующим образом:

```
vagrant@jessie:~$ pwd
/home/vagrant
vagrant@jessie:~$
```

Команда pwd просто выводит полное имя каталога, в котором вы находитесь в текущий момент, то есть имя рабочего каталога.

Теперь вы знаете свое местоположение в файловой системе и можете начать перемещение по ней с помощью команды cd (change directory), указывая путь к цели. Например, если вы находитесь в своем домашнем каталоге и хотите перейти в подкаталог *bin*, то надо просто ввести команду cd bin и нажать клавишу **Enter** (или **Return**).

Внимание: обратите внимание на отсутствие символа слеша перед именем подкаталога в приведенной выше команде. Дело в том, что в большинстве случаев */bin* и *bin* являются абсолютно разными каталогами в файловой системе (как правило, у пользователя есть собственный локальный подкаталог *bin*):

Tlgm: @it_boooks

- использование `bin` (без предшествующего слеша) означает переход в подкаталог *bin*, расположенный в текущем рабочем каталоге;
- использование `/bin` (с предшествующим слешем) означает переход в подкаталог *bin*, расположенный в корневом (/) каталоге.

Итак, мы выяснили, что *bin* и */bin* могут быть совершенно разными локациями. Именно поэтому столь важно полное понимание концепции файловой системы с единственным корнем и полного путевого имени к файлу или каталогу. В противном случае в конечном итоге вы можете обнаружить, что выполнили действие совсем не с тем файлом или каталогом, с которым намеревались. Это особенно важно понимать при работе непосредственно с файлами и каталогами, которая будет подробно описана в следующем разделе.

Но прежде чем продолжить, рассмотрим еще несколько полезных команд перемещения, заслуживающих внимания.

Для перемещения на один уровень вверх по файловой системе (например, из каталога */usr/local/bin* в каталог */usr/local*) можно воспользоваться сокращенным обозначением .. (две точки). Каждый каталог содержит особый элемент, обозначенный двумя точками (..), соответствующий родительскому каталогу, то есть каталогу, расположенному на один уровень выше текущего. Если текущим каталогом является */usr/local/bin*, то просто введите команду `cd ..` и нажмите клавишу **Enter** (или **Return**), чтобы перейти в каталог уровнем выше (в родительский каталог):

```
vagrant@jessie:/usr/local/bin$ cd ..
vagrant@jessie:/usr/local$
```

Следует отметить, что сокращенное обозначение .. можно объединять с именем каталога, чтобы перемещаться в соседние каталоги, размещенные на одном уровне. Например, если текущим каталогом является */usr/local* и нужно перейти в каталог */usr/share*, то можно ввести команду `cd ../share` и нажать клавишу **Enter**. Результатом будет переход в каталог с именем *share*, чей путь начинается одним уровнем выше.

```
vagrant@jessie:/usr/local$ cd ../share
vagrant@jessie:/usr/share$
```

Кроме того, можно указывать несколько уровней с помощью сокращения .. для перемещения сразу на несколько уровней вверх. Например, если текущим каталогом является */usr/share* и нужно перейти в корневой каталог (/), то можно ввести команду `cd ../..` и нажать клавишу **Enter**. Результатом будет переход в корневой каталог.

```
vagrant@jessie:/usr/share$ cd ../..
vagrant@jessie:/$
```

Во всех приведенных примерах используется относительный путь (relative path), то есть путь, определяемый относительно текущей локации. Разумеется, можно также использовать абсолютный путь (absolute path) – путь, который всегда определяется по отношению к корневому каталогу. Как уже было

Tlgm: @it_boooks

отмечено выше, различие состоит в использовании начального слеша (/) для обозначения абсолютного пути, начинающегося с корневого каталога. Если начальный слеш отсутствует, то путь определяется относительно текущего каталога. Например, если вы находитесь в корневом каталоге и хотите перейти в каталог */media/cdrom*, то нет необходимости в указании начального слеша (так как *media* – это подкаталог корневого каталога). Вы можете ввести команду `cd media/cdrom` и нажать клавишу **Enter**. В результате вы переместитесь в подкаталог */media/cdrom*, поскольку воспользовались относительным путем к цели.

```
vagrant@jessie:/$ cd media/cdrom
vagrant@jessie:/media/cdrom$
```

Но если из этого подкаталога необходимо перейти в подкаталог */usr/local/bin*, то, вероятнее всего, придется использовать абсолютный путь. Почему? Потому что между этими двумя локациями не существует (простого) относительного пути, который не проходил бы через корневой каталог (более подробно см. примечание «Существует несколько путей» ниже). Использование абсолютного пути с указанием начального слеша – это самый быстрый и простой способ перемещения между каталогами.

```
vagrant@jessie:/media/cdrom$ cd /usr/local/bin
vagrant@jessie:/usr/local/bin$
```



Существует несколько путей

Если вы догадались, что можно воспользоваться командой `cd ../../usr/local/bin` для перехода из каталога /media/cdrom в каталог /usr/local/bin, то вы в полной мере поняли взаимотношения между относительными и абсолютными путями в Linux-системе.

В заключение рассмотрим один неочевидный способ перемещения по файловой системе. Предположим, что вы находитесь в каталоге */usr/local/bin* и вам нужно перейти в каталог */media/cdrom*. Вы вводите команду `cd /media/cdrom`, но после перехода понимаете, что вам все-таки нужно было остаться в каталоге */usr/local/bin*. К счастью, это можно сделать легко и быстро. Форма команды `cd -` (использование дефиса как аргумента команды `cd`) сообщает командной оболочке bash о необходимости возврата в каталог, в котором вы находились перед переходом в текущий каталог. А если необходимо вернуться в свой домашний каталог, просто введите команду `cd` без параметров.

```
vagrant@jessie:/usr/local/bin$ cd /media/cdrom
vagrant@jessie:/media/cdrom$ cd -
/usr/local/bin
vagrant@jessie:/usr/local/bin$ cd -
/media/cdrom
vagrant@jessie:/media/cdrom$ cd -
/usr/local/bin
vagrant@jessie:/usr/local/bin$
```

А сейчас продемонстрируем все рассмотренные выше способы перемещения по файловой системе:

Tlgm: @it_boooks


```
vagrant@jessie:/usr/local/bin$ cd ..
vagrant@jessie:/usr/local$ cd ../share
vagrant@jessie:/usr/share$ cd ../..
vagrant@jessie:/$ cd media/cdrom
vagrant@jessie:/media/cdrom$ cd /usr/local/bin
vagrant@jessie:/usr/local/bin$ cd -
/media/cdrom
vagrant@jessie:/media/cdrom$ cd -
/usr/local/bin
vagrant@jessie:/usr/local/bin$
```

Теперь вы имеете достаточно полное представление о методиках перемещения по файловой системе ОС Linux. Пополним наши знания информацией о работе с файлами и каталогами.

Работа с файлами и каталогами

После получения базовых знаний о файловой системе Linux, о путях к файлам и о способах перемещения по файловой системе переходим к изучению работы с файлами и каталогами. Здесь будут рассматриваться четыре основные темы:

- создание файлов и каталогов;
- удаление файлов и каталогов;
- перемещение, копирование и переименование файлов и каталогов;
- изменение прав доступа.

Начнем с изучения способов создания файлов и каталогов.

Создание файлов и каталогов

Для создания файлов или каталогов предназначены две основные команды: `touch`, используемая для создания файлов, и `mkdir` (make directory) – из названия понятно, что эта команда создает каталоги.



Другие способы создания файлов

Существуют и другие способы создания файлов, такие как перенаправление потока вывода команды в файл или использование приложения (например, текстового редактора). Мы не будем пытаться охватить все возможные способы выполнения каких-либо действий (в нашем случае – создания файлов), нам необходимо сосредоточиться на получении базовой информации, достаточной для того, чтобы начать работу.

Команда `touch` просто создает новый файл без содержимого (вам предоставляется возможность воспользоваться текстовым редактором или другим подходящим приложением для добавления содержимого в этот файл после его создания). Рассмотрим несколько примеров:

```
[vagrant@centos ~]$ touch config.txt
```

Команда, равнозначная по смыслу (ниже будет объяснено, почему эти команды равнозначны):

```
[vagrant@centos ~]$ touch ./config.txt
```

Tlgm: @it_boooks

Вероятно, не сразу можно понять, почему эти две команды равнозначны по смыслу (и по результату). В предыдущем разделе было описано сокращенное обозначение .. (две точки) для перемещения в родительский каталог из текущего каталога. Но, кроме того, каждый каталог содержит запись, обозначенную одной точкой (.), которая указывает на текущий каталог (current directory). Таким образом, обе команды `touch config.txt` и `touch ./config.txt` создают файл с именем *config.txt* в текущем каталоге.

Если обе команды с точки зрения синтаксиса корректны, то зачем нужны два различных способа выполнения одного и того же действия? В рассматриваемом выше случае обе команды дают одинаковый результат, но это справедливо не для всех команд. Если вам нужна полная уверенность в том, что файл будет создан в текущем рабочем каталоге, то необходимо использовать префикс `./`, чтобы точно указать командной оболочке текущий каталог как место расположения создаваемого файла.

```
[vagrant@centos ~]$ touch /config.txt
```

В этом примере используется абсолютный путь, поэтому команда создает файл с именем *config.txt* в корневом каталоге, если ваша учетная пользовательская запись имеет на это право. (Более подробно об этом см. ниже, в разделе «Изменение прав доступа».)



В каких случаях полезен префикс ./

Мы пока еще не рассматривали подробно концепцию путей поиска (search paths), реализованную в командной оболочке bash. Пути поиска – это специально определенные пути (локации) в файловой системе, в которых bash автоматически выполняет поиск, когда пользователь вводит какую-либо команду. В обычной конфигурации в путь поиска включаются /bin, /usr/bin, /sbin и прочие подобные локации. Таким образом, при вводе имени файла, расположенного в одном из этих каталогов, без указания полного пути к нему bash найдет его сам, выполняя поиск по этим путям. Это один из тех случаев, когда точное указание места расположения файла (с помощью префикса *./* или абсолютного пути) может стать необходимым, если вы хотите точно указать, какой именно файл должна найти командная оболочка для выполнения заданной команды.

Команда `mkdir` очень проста: она создает каталог с именем, заданным пользователем. Рассмотрим несколько несложных примеров:

```
[vagrant@centos ~]$ mkdir bin
```

Эта команда создает каталог с именем *bin* в текущем рабочем каталоге. Приведенная ниже команда дает совершенно другой результат (помните о различии относительного и абсолютного путей!):

```
[vagrant@centos ~]$ mkdir /bin
```

Подобно большинству команд Linux, команда `mkdir` имеет большое количество ключей и опций, изменяющих ее поведение, но достаточно часто вы будете использовать ключ `-p`. При выполнении команды с ключом `-p` `mkdir` не

Tlgm: @it_boooks

выводит сообщение об ошибке, если заданный каталог уже существует, и продолжит создание последовательности вложенных каталогов по указанному пути при необходимости.

Например, предположим, что необходимо сохранить несколько файлов в подкаталоге */opt/sw/network*. Если вы находитесь в каталоге */opt* и выполняете команду `mkdir sw/network`, но каталога *sw* пока еще не существует, то команда `mkdir` выведет сообщение об ошибке и прервет выполнение. Но если добавить ключ `-p`, то `mkdir` создаст каталог *sw*, если это необходимо, затем создаст в каталоге *sw* подкаталог *network*. Это дает замечательную возможность создания всей необходимой цепочки вложенных подкаталогов одной командой независимо от существования или отсутствия каталогов в заданной последовательности и без получения сообщений об ошибках.

Создание файлов и каталогов – это только первая часть нашей учебной задачи, рассмотрим вторую часть – удаление файлов и каталогов.

Удаление файлов и каталогов

В предыдущем разделе мы выяснили, что для создания файлов и каталогов существуют две основные команды. Точно так же для удаления файлов и каталогов существуют две основные команды. В большинстве случаев команда `rm` (remove) используется для удаления файлов, а команда `rmdir` – для удаления каталогов. Тем не менее есть способ применения команды `rm` для удаления каталогов, и мы рассмотрим этот способ в текущем разделе.

Формат команды удаления файла прост: `rm` *`filename`*. Например, для удаления файла *config.txt* из текущего рабочего каталога можно воспользоваться одной из двух следующих команд (вам понятно, почему?):

```
vagrant@trusty:~$ rm config.txt
vagrant@trusty:~$ rm ./config.txt
```

Разумеется, можно использовать и абсолютные пути (`/home/vagrant/config.txt`), и относительные пути (`./config.txt`).

Для удаления каталога применяется команда `rmdir` *`directory`*. Важно отметить, что удаляемый каталог обязательно должен быть пустым. Если вы попытаетесь удалить каталог, в котором есть файлы, то получите следующее сообщение об ошибке:

```
rmdir: failed to remove 'src': Directory not empty
(rmdir: невозможно удалить 'src': Каталог не пуст)
```

В этом случае сначала нужно очистить каталог (то есть удалить в нем все файлы), затем выполнить команду `rmdir`. Есть и другой способ: воспользоваться ключом `-r` команды `rm`. Обычно при попытке применения команды `rm` для удаления каталога без ключа `-r` командная оболочка выдает сообщение, похожее на приведенное ниже (в этом примере была сделана попытка удалить каталог *bin* в текущем рабочем каталоге):

```
rm: cannot remove 'bin': Is a directory
(rm: невозможно удалить 'bin': Это каталог)
```

Tlgm: @it_boooks

Но при использовании команды `rm -r` *`directory`* командная оболочка удалит все поддерево каталогов. Важно отметить, что по умолчанию команда `rm` не запрашивает подтверждение – она просто удаляет все указанное поддерево каталогов. Никаких Корзин (Recycle Bin, Trash Can и т. п.) – все исчезает бесследно. Если необходим запрос на подтверждение удаления, можно добавить ключ `-i`.



Сказанное выше также относится к командам `mv` и `cp`, которые будут рассматриваться в следующем разделе, – без ключа `-i` эти команды «молча» перезаписывают файлы в целевой локации без запросов и предупреждений. Поэтому при использовании данных команд необходима достаточно высокая степень внимательности и осторожности.

Но наши задачи не ограничиваются только созданием и удалением файлов и каталогов, поэтому переходим к операциям копирования и перемещения файлов и каталогов.

Перемещение, копирование и переименование файлов и каталогов

Для перемещения, копирования и переименования файлов и каталогов предназначены две команды: `cp` – для копирования файлов и каталогов и `mv` – для перемещения и переименования файлов и каталогов.



Изучайте страницы руководства man

Демонстрируемые до сих пор основные способы использования команд Linux были достаточно простыми для понимания, но, как говорится, «дьявол кроется в деталях (мелких подробностях)». Для получения более подробной информации о любом ключе, опции и параметре или для понимания всех нюансов более хитроумного применения всех (почти всех) команд Linux как можно чаще пользуйтесь командой `man` (manual). Например, для чтения страницы руководства по использованию команды `cp` введите в командной строке `man cp`. Страницы руководства подробнейшим образом описывают использование разнообразных команд.

Для копирования файла выполняется команда `cp` *`source destination`*. Для перемещения файла предназначена команда `mv` *`source destination`*. Операция переименования файла рассматривается как перемещение файла со сменой имени (обычно в одном и том же каталоге).

Перемещение каталога выполняется практически так же: `mv` *`source-dir destination-dir`*. Команда правильно работает, если исходный каталог простой (то есть содержит только файлы) и если исходный каталог представляет собой поддерево (то есть содержит и файлы, и подкаталоги).

Копирование каталогов является несколько более сложной процедурой. Необходимо добавить ключ `-r`, то есть шаблон команды выглядит так: `cp -r` *`source-dir destination-dir`*. Этого достаточно в большинстве случаев для правильного копирования каталогов, хотя в некоторых особых случаях могут потребоваться дополнительные ключи (но такие случаи встречаются гораздо реже). Рекомендуется внимательно изучить страницу руководства man по использованию команды `cp` и постоянно обращаться к ней, чтобы получить полную информацию

Tlgm: @it_boooks

о нюансах использования этой команды (см. примечание «Изучайте страницы руководства man» выше).

Последняя тема данного учебного раздела – права доступа к файлам и каталогам.

Изменение прав доступа

Как и все Unix-предшественники (напомним, что Linux стал плодом усилий по созданию свободной Unix-подобной операционной системы), Linux представляет собой многопользовательскую ОС, в которой применяется механизм прав доступа к файлам и каталогам. Для признания полноценной многопользовательской операционной системой в Linux необходимо было сформировать механизм, гарантирующий, что пользователь не получит возможности отслеживать, просматривать содержимое, изменять и удалять файлы (и каталоги) другого пользователя, поэтому установка прав доступа на уровне файлов и каталогов стала практической необходимостью.

Механизм прав доступа в Linux основан на двух главных концепциях:

- права доступа назначаются на уровне пользователя (пользователя, который является владельцем файла), группы (других пользователей, принадлежащих к группе пользователя-владельца) и всех остальных (пользователей, не входящих в группу пользователя-владельца);
- права доступа основаны на возможности совершения действий (чтение, запись, выполнение).

Теперь рассмотрим, как объединяются эти концепции в практической реализации. Каждому из перечисленных выше действий (чтение, запись, выполнение) присваивается числовое значение: 4 – чтение, 2 – запись, 1 – выполнение. (Обратите внимание на то, что эти числовые значения в бинарной форме представляют собой единицы в трех последовательных разрядах: 4 = 100, 2 = 010, 1 = 001.) Чтобы разрешить несколько действий, нужно просуммировать соответствующие значения для каждого действия. Например, для предоставления прав записи и чтения в сумме получается 6 (чтение = 4, запись = 2, следовательно, чтение + запись = 6).

Полученные суммарные значения присваиваются пользователю-владельцу, группе и всем остальным. Например, чтобы разрешить владельцу чтение и запись собственного файла, нужно установить для прав доступа владельца значение 6. Если владельцу требуется чтение, запись и выполнение файла, то для прав доступа владельца устанавливается значение 7. Аналогичным образом устанавливаются значения для прав доступа пользователям из группы владельца файла: если необходимо, чтобы группа получила возможность чтения файла, но с запрещением его записи и выполнения, то для прав доступа группы устанавливается значение 4. Права доступа владельца, группы и прочих пользователей интерпретируются как восьмеричные числа:

```
644 (владелец = чтение+запись, группа = чтение, прочие = чтение)
755 (владелец = чтение+запись+выполнение, группа = чтение+выполнение, прочие =
чтение+выполнение)
```

Tlgm: @it_boooks


```
600 (владелец = чтение+запись, группа = нет, прочие = нет)
620 (владелец = чтение+запись, группа = запись, прочие = нет)
```

Кроме того, эти же права доступа можно видеть в форме строки символов, например `rwxr-xr-x`. Здесь права доступа обозначены соответствующими буквами: чтение – r (read), запись – w (write), выполнение – x (execution) для каждого из трех объектов (владелец, группа, прочие). Примеры записи прав доступа, приведенные выше, можно записать в символьном формате:

```
644 = rw-r--r--
755 = rwxr-xr-x
600 = rw-------
620 = rw--w----
```

Права на чтение и запись вряд ли требуют каких-либо разъяснений, но право на выполнение немного отличается от них. Для файла право на выполнение полностью соответствует названию: это разрешение выполнить файл как программу (более подробно об этом см. в следующем разделе «Выполнение программ»). Но для каталога право на выполнение означает возможность просматривать каталог и выводить список его содержимого. Таким образом, если пользователям из группы владельца каталога нужно разрешить просмотр его содержимого, необходимо установить для них право на выполнение.

Для просмотра и изменения прав доступа в Linux применяется несколько системных утилит. Вероятно, чаще всего вы будете пользоваться утилитой `ls`, выводящей содержимое каталога, а также права доступа (и некоторую другую информацию) при использовании ключа `-l`. На рис. 3.1 показан вывод команды `ls -l /bin` в системе Debian 8.1, где в первом столбце ясно видны права доступа, присвоенные файлам в этом списке.

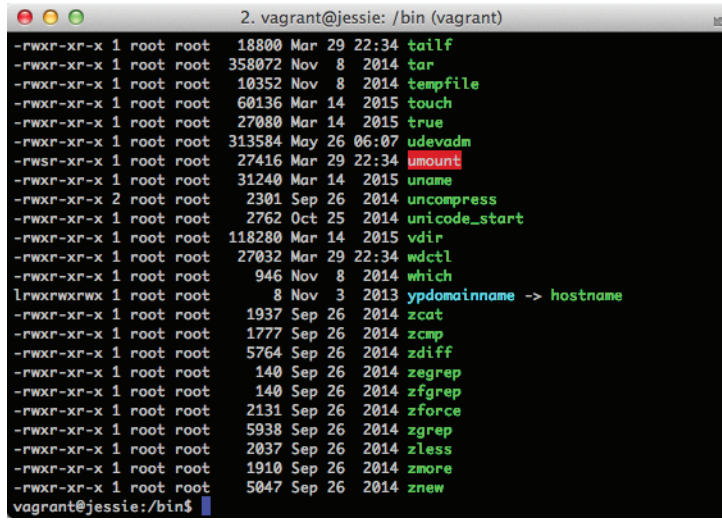


Рис. 3.1 ❖ Вывод прав доступа в списке файлов

Tlgm: @it_boooks

Для назначения или изменения прав доступа потребуется утилита chmod. Для нее применяется и восьмеричная форма записи (755, 600, 644 и т. д.), и форма строки символов rwxr-xr-x (ее обычно называют символьной нотацией), поскольку chmod «понимает» обе формы записи прав доступа. Как и при сравнении применения относительного и абсолютного путей, при выборе использования восьмеричного формата или символьной нотации прав доступа можно дать следующие рекомендации:

- если необходимо одновременно установить (или изменить) все права доступа, то следует использовать восьмеричные значения. Даже если вы пропустите одну из цифр, ошибки не будет, потому что chmod при недостаточном количестве цифр добавляет начальные нули (ноль означает отсутствие всех трех прав доступа), поэтому команда в любом случае будет выполнена;
- если необходимо установить только одну часть прав доступа (для владельца, для группы или для прочих), а все остальные части оставить без изменений, то следует использовать символическую нотацию. Команда chmod позволяет выполнить такое выборочное изменение (например, только для владельца или только для группы)[1].

Ниже приведено несколько простых примеров применения утилиты chmod. Сначала для подкаталога *bin* в текущем рабочем каталоге устанавливаются права доступа 755 (владелец = чтение/запись/выполнение, все прочие = чтение/выполнение):

```
[vagrant@centos ~]$ chmod 755 bin
```

Далее используется символьная нотация для добавления права чтения/записи владельцу файла *config.txt* в текущем рабочем каталоге, при этом все другие права доступа остаются неизменными:

```
[vagrant@centos ~]$ chmod u+rw config.txt
```

Вот немного более сложный пример – добавление прав чтения/записи для владельца файла и удаление права на запись для группы:

```
[vagrant@centos ~]$ chmod u+rw,g-w /opt/share/config.txt
```

Кроме того, в команде chmod можно использовать ключ -R для рекурсивного выполнения, то есть для распространения заданного изменения прав доступа на все файлы и подкаталоги нижележащих уровней (разумеется, этот ключ работает, когда команда chmod применяется к каталогу).

---

[1] Вообще говоря, эти рекомендации нельзя назвать универсальными. Все зависит от индивидуального восприятия форматов записи прав доступа – кому-то проще разобраться с цифровыми значениями, кто-то предпочтет символьную нотацию. В любом случае, проще будет изменять права доступа тем способом, который воспринимается как более понятный и эффективный. – *Прим. перев.*

Tlgm: @it_boooks

Изменение владельца и группы

Поскольку владелец и группа играют важную роль при определении прав доступа к файлу, вполне естественно, что в ОС Linux имеются инструментальные средства для изменения владельца и группы, применяемые к файлам и каталогам (для просмотра владельца и группы также используется команда `ls`, как показано на рис. 3.1). Команда `chown` позволяет изменить владельца файла, команда `chgrp` изменяет группу. Обе команды поддерживают ключ `-R` для рекурсивного выполнения по аналогии с командой `chmod`.

После изучения основ работы с файлами и каталогами переходим к следующей важной теме – выполнение программ в ОС Linux.

Выполнение программ

Выполнение программ в ОС Linux осуществляется достаточно просто, если принять во внимание ранее изученный материал. Для того чтобы запустить (выполнить) программу, необходимы следующие условия:

- файл действительно должен быть выполняемым (executable). Чтобы определить, является ли файл выполняемым, воспользуйтесь утилитой `file`;
- для файла должно быть установлено право на выполнение (либо для владельца, либо для члена группы владельца, либо для всех остальных).

Второе условие (право на выполнение) было описано в предыдущем разделе, поэтому здесь мы не будем повторяться. Если у вас недостаточно прав для выполнения файла, используйте команды `chmod`, `chown`, `chgrp`, чтобы соответствовать второму условию. Первое требование необходимо рассмотреть более подробно.

Что означает термин «выполняемый файл»? Возможно, это бинарный файл, скомпилированный из исходных кодов, написанных на одном из языков программирования, например C или C++. Выполняемым также может быть и текстовый файл, например скрипт на языке командной оболочки bash (последовательность команд bash) или скрипт на интерпретируемом языке программирования, например Python или Ruby. (В следующей главе мы рассмотрим язык Python весьма подробно.) В определении выполняемости файла может помочь утилита `file` (обычно она устанавливается по умолчанию, но могут быть исключения; если утилита отсутствует, воспользуйтесь менеджером пакетов вашего дистрибутива, чтобы установить ее).

Ниже приведены примеры вывода команды `file` для различных типов выполняемых файлов:

```
vagrant@jessie:~$ file /bin/bash
/bin/bash: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=a8ff57737fe60fba639d91d603253f4cdc6eb9f7, stripped
vagrant@jessie:~$ file docker
docker: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.24,
BuildID[sha1]=3d4e8c5339180d462a7f43e62ede4f231d625f71, not stripped
```

Tlgm: @it_boooks


```
vagrant@jessie:~$ file shellscript.sh
script.sh: Bourne-Again shell script, ASCII text executable
vagrant@jessie:~$ file testscript.py
script.py: Python script, ASCII text executable
vagrant@jessie:~$ file testscript-2.rb
script.rb: Ruby script, ASCII text executable
```



Скрипты и shebang

Обратите внимание на замечательную способность команды `file` определять текстовые файлы как скрипт на языке Python, скрипт на языке Ruby или скрипт на языке командной оболочки bash. Это может показаться маленьким чудом, но в действительности точное определение возможно благодаря специальной конструкции, известной под названием shebang. Shebang – это первая строка в текстовом скрипте, которая начинается с комбинации символов #!, за которой следует путь к интерпретатору данного скрипта (интерпретатор (interpreter) – это программа, которая будет выполнять команды, записанные в скрипте). Например, в системе Debian 8.1 интерпретатор Python расположен в локации /usr/bin/python, поэтому конструкция shebang для скрипта на языке Python должна выглядеть следующим образом: `#!/usr/bin/python`. Для скрипта на языке Ruby конструкция shebang аналогична, но указывает на интерпретатор Ruby. Shebang в скриптах на bash указывает на саму командную оболочку bash, разумеется[1].

После удовлетворения обоих условий – имеется выполняемый файл, и пользователь обладает правом на выполнение этого файла – запуск программы осуществляется очень просто: имя файла программы/скрипта вводится в командной строке, после чего нажимается клавиша **Enter**. Разумеется, каждая программа обладает собственным набором ключей, опций и параметров, которые могут потребоваться при запуске. Необходимо выяснить один вопрос, касающийся использования абсолютных путей: например, если в системе существует несколько программ с именем `testnet` и в командной строке просто вводится имя `testnet`, то какая из программ будет выполнена? Здесь требуется полное понимание концепции путей поиска в командной оболочке bash (см. ниже). В этом случае выполнить именно ту программу, которая вам нужна, помогает указание полного абсолютного пути к ней.

Рассмотрим этот вопрос немного подробнее. В начале этой главы, в разделе «Перемещение по файловой системе», обсуждалась концепция относительных и абсолютных путей. Теперь в продолжение этого обсуждения введем концепцию пути поиска (search path) . В каждой Linux-системе определен путь поиска, представляющий собой список каталогов системы, в которых будет производиться поиск, когда пользователь вводит имя файла в командной строке. Текущий установленный путь поиска можно посмотреть с помощью команды `echo $PATH`. В системе CentOS путь поиска может выглядеть следующим образом:

---

[1] В русскоязычной литературе термин shebang встречается крайне редко, в Википедии используется калька «шебанг» (https://ru.wikipedia.org/wiki/Шебанг_(Unix)). – *Прим. перев.*

Tlgm: @it_boooks


```
[vagrant@centos ~]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/vagrant/.local/bin:/home/vagrant/bin
[vagrant@centos ~]$
```

Таким образом, если скрипт *testscript.py* содержится в каталоге */usr/local/bin*, то вы можете находиться в любом каталоге файловой системы, но при вводе в командной строке имени *testscript.py* будет выполнен именно этот скрипт. Система просматривает каталоги, указанные в пути поиска (в том порядке, в котором они записаны), чтобы найти введенное имя файла, и выполняет первый найденный файл с этим именем (в нашем примере это файл в каталоге */usr/local/bin*, то есть в самом первом каталоге в пути поиска).

Следует обратить особое внимание на то, что в путь поиска не включен текущий каталог. Предположим, что вы создали каталог *scripts* в своем домашнем каталоге, и в этом каталоге имеется скрипт с именем *shellscript.sh*. Рассмотрим поведение системы при выполнении следующих команд:

```
[vagrant@centos ~]$ pwd
/home/vagrant/scripts
[vagrant@centos ~]$ ls
shellscript.sh
[vagrant@centos ~]$ shellscript.sh
-bash: /home/vagrant/bin/shellscript.sh: No such file or directory (Нет такого файла или
каталога)
[vagrant@centos ~]$ ./shellscript.sh
This is a shell script.
[vagrant@centos ~]$
```

Наш скрипт не был обнаружен в пути поиска, поэтому мы использовали абсолютный путь – в данном примере это абсолютный путь, указывающий на необходимость выполнения файла из текущего каталога (с помощью нотации `./`).

Таким образом, концепция выполнения программ состоит в том, что любая запускаемая программа – скомпилированный бинарный файл или текстовый скрипт, интерпретируемый командной оболочкой, Python, Ruby или любым другим интерпретатором, – обязательно должна находиться в пути поиска. В противном случае необходимо явно указать абсолютный путь (который может включать текущий каталог) к нужной программе. Если имеется несколько программ с одинаковым именем в различных каталогах, то командная оболочка выполнит тот файл, который найдет первым, а порядок просмотра каталогов определяется путем поиска.

Чтобы узнать, существует ли несколько программ с одинаковым именем, можно воспользоваться командой `which`. Например, предположим, что вы написали на языке Python скрипт с именем `uptime`, собирающий статистические данные о непрерывном времени работы сетевых устройств. В большинстве дистрибутивов Linux имеется системная команда `uptime` (она выводит информацию о времени непрерывной работы текущей системы). С помощью команды `which uptime` вы получаете полный путь к самому первому найденному в пути поиска выполняемому файлу с именем `uptime`. (Это та программа, кото-

Tlgm: @it_boooks

рая была бы выполнена при вводе в командной строке имени `uptime`.) Получив эту информацию, вы понимаете, что необходимо указать полный путь к вашему скрипту на языке Python или дополнить/изменить путь поиска (если это действительно необходимо).



Разумеется, вы можете изменять и настраивать путь поиска по своему усмотрению. Путь поиска управляется переменной среды (environment variable) с именем `PATH`. (По неофициальному соглашению имена всех переменных среды записываются буквами верхнего регистра.) Изменение значения этой переменной среды меняет порядок просмотра каталогов при поиске выполняемых программ.

До перехода к подробному обсуждению поддержки сетей в ОС Linux необходимо рассмотреть работу с программами, которые всегда выполняются в фоновом режиме. Такие программы в Unix-подобных ОС традиционно называют демонами.

Работа с демонами

В мире Linux термин «демон» (daemon) обозначает процесс, выполняемый в фоновом режиме. (Иногда для обозначения этого типа фоновых процессов используют термин «сервис» (service).) Чаще всего придется иметь дело с демонами, если Linux используется для обеспечения сетевой функциональности. С некоторыми примерами демонов мы уже встречались в самом первом разделе, описывающем взаимодействие с ОС Linux, – сервер DHCP, сервер HTTP, сервер DNS, сервер FTP. В Linux-системе каждый из этих сетевых сервисов представлен соответствующим демоном (или системным сервисом). В этом разделе мы рассмотрим основные методы работы с демонами: запуск демонов, останов демонов, перезапуск демона и проверку состояния (статуса) демона.

Обычно работа с демонами в Linux-системе существенно отличается для различных дистрибутивов. Ранее для запуска, останова и перезапуска демонов использовались специальные стартовые (startup) скрипты, или init-скрипты. Некоторые дистрибутивы стали предлагать утилиты (часто являющиеся теми же bash-скриптами), такие как команда `service`, чтобы упростить работу с демонами. Например, в системах Ubuntu 14.04 LTS и CentOS 7.1 команда `service` (расположенная в каталоге */usr/sbin*) позволяет пользователю запускать, останавливать и перезапускать демона. Внутри таких утилит вызывались команды, специфические для конкретного дистрибутива (например, `initctl` в Ubuntu или `systemctl` в CentOS), для действительного выполнения этих действий.

Но в последние годы подавляющее большинство дистрибутивов Linux перешло на использование systemd в качестве системы инициализации (init system): RHEL/CentOS 7.x Debian 8.0 и более поздние версии, а также Ubuntu 15.04 и более поздние версии полностью перешли на systemd.



Если вас интересует более подробная информация о systemd, то рекомендуем посетить веб-сайт http://freedesktop.org/wiki/Software/systemd.

Tlgm: @it_boooks

Мы будем рассматривать работу с демонами в трех основных дистрибутивах Linux, выбранных для этой главы: Debian GNU/Linux 8.1 («Jessie»), Ubuntu «Trusty Tahr» 14.04 LTS и CentOS 7.1. Начнем с Debian GNU/Linux 8.1.



Systemd заслуживает большего внимания

Обсуждение systemd требует большего внимания, чем мы можем уделить здесь. В примерах, демонстрирующих запуск, останов и перезапуск фонового сервиса с использованием systemd, предполагается, что unit-файл systemd уже установлен, настроен и активизирован, следовательно, распознается systemd.

Работа с фоновыми сервисами в Debian GNU/Linux 8.1

В Debian GNU/Linux systemd как система инициализации (init system) используется с версии 8.0, поэтому доступ к основным инструментам для работы с фоновыми сервисами осуществляется через утилиту `systemctl` (размещена в каталоге */bin/systemctl*). В отличие от некоторых других дистрибутивов, Debian не использует команд-«оберток», вызывающих `systemctl` скрыто от пользователя, а предлагает пользоваться `systemctl` напрямую.

Для запуска демона с помощью systemd необходимо выполнить команду `systemctl` с подкомандой `start` (здесь термин «подкоманда» (subcommand) используется для обозначения параметра, передаваемого `systemctl`, как действие, которое утилита должна выполнить, – это обозначение мы будем использовать и далее в данной главе при рассмотрении работы с сетью в Linux):

```
vagrant@jessie:~$ systemctl start имя-сервиса
```

Для останова демона с помощью systemd подкоманда `start` заменяется на `stop`, как в следующем примере:

```
vagrant@jessie:~$ systemctl stop имя-сервиса
```

Аналогичным образом задается подкоманда `restart` для останова и последующего запуска демона:

```
vagrant@jessie:~$ systemctl restart имя-сервиса
```

Отметим, что `systemctl` также поддерживает подкоманду `reload`, выполняющую перезагрузку конфигурации демона. Это действие может оказаться менее опасным для нормальной работы системы, чем перезапуск демона (с помощью команды `systemctl restart`, которая почти всегда потенциально опасна), но поведение демона в конкретных случаях после перезагрузки конфигурации будет различным (другими словами, не все демоны способны автоматически адаптироваться к новой конфигурации и сохранить нормальный режим работы).

Подкоманда `status` утилиты `systemctl` предназначена для проверки текущего состояния демона. На рис. 3.2 показан вывод команды `systemctl status` в виртуальной машине Debian 8.1.

Tlgm: @it_boooks

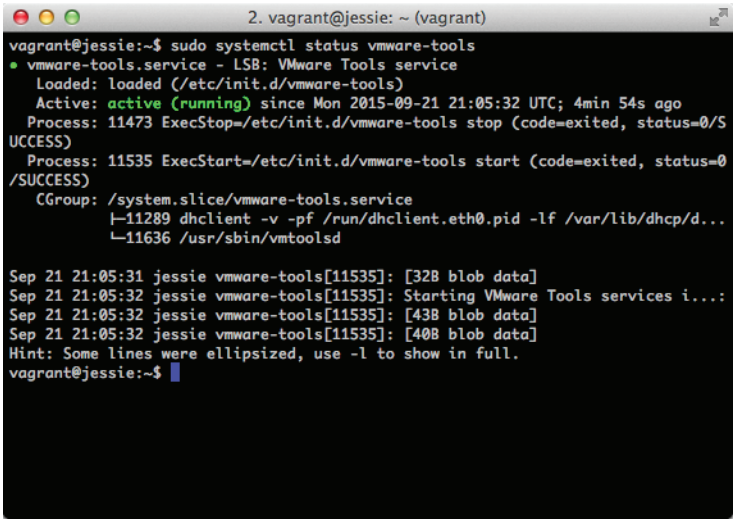


Рис. 3.2 ❖ Вывод команды systemctl status

Если необходимо узнать имя сервиса, то команда `systemctl list-units` поможет получить постранично выводимый список всех загруженных и активных юнитов (units).



До версии 8.0 в Debian не использовалась система инициализации systemd. Вместо нее применялась более старая система инициализации под названием System V init или sysv-rc.

Теперь перейдем к рассмотрению работы с демонами в дистрибутиве Ubuntu Linux 14.04 LTS. Несмотря на то что Ubuntu Linux является дистрибутивом, производным от Debian, вы увидите, что имеются существенные различия между Debian 8.x и этим LTS-релизом Ubuntu.

Работа с фоновыми сервисами в Ubuntu Linux 14.04 LTS

В отличие от Debian 8.x и CentOS 7.x, Ubuntu 14.04 LTS (напомним, что LTS обозначает релиз с долговременной поддержкой (long-term support) – в течение пяти лет с момента выпуска) не использует systemd как систему инициализации. Вместо нее в Ubuntu 14.04 применяется разработанная в компании Canonical система Upstart. (В следующем основном LTS-релизе Ubuntu 16.04 будет использована systemd. Здесь мы рассматриваем версию 14.04, потому что во многих реально эксплуатируемых производственных средах с большой вероятностью используется именно эта LTS-версия.)

Основной командой, используемой для взаимодействия с Upstart для запуска, останова, перезапуска и проверки состояния фоновых сервисов (в терминологии Upstart они обозначаются как «задания» или «службы» – jobs), является `initctl`, работа с которой очень похожа на работу с утилитой `systemctl`.

Tlgm: @it_boooks

Например, для запуска демона используется следующая команда:

```
vagrant@trusty:~$ initctl start имя-сервиса
service name start/running
```

Пример команды для останова демона:

```
vagrant@trusty:~$ initctl stop имя-севиса
service name stop/waiting
```

Подкоманды `restart` и `status` работают почти аналогичным образом. Ниже показаны примеры перезапуска и проверки состояния демона VMware Tools (VMware Tools – это фоновый сервис, часто устанавливаемый в виртуальных машинах на основе VMware):

```
vagrant@trusty:~$ initctl restart vmware-tools
vmware-tools start/running
vagrant@trusty:~$ initctl status vmware-tools
vmware-tools start/running
```

Как и при использовании `systemctl`, здесь имеется возможность вывода списка имен сервисов, с помощью которого можно узнать точное имя сервиса, которое необходимо указать при запуске, останове или проверке состояния демона:

```
vagrant@trusty:~$ initctl list
```

Кроме того, в Ubuntu 14.04 LTS поддерживаются сокращенные обозначения некоторых команд для работы с демонами:

- команды `start`, `stop`, `restart` и `status` являются символическими ссылками на команду (файл) initctl. Каждая из этих команд работает так, как если бы вы ввели полную команду `initctl` *subcommand*, например `stop vmware-tools` работает точно так же, как `initctl stop vmware-tools`. Эти символические ссылки размещены в каталоге */sbin*;
- в Ubuntu также имеется скрипт командной оболочки с именем `service`, в котором скрыто вызывается `initctl`. Формат этой команды: `service service subcommand`, где `service` – имя демона (которое можно узнать, выполнив команду `initctl list`), `subcommand` – одна из подкоманд `start`, `stop`, `restart` или `status`. Отметим, что синтаксис скрипта противоположен синтаксису самой команды `initctl`, формат которой: `initctl` *subcommand service*. Это может привести к некоторой путанице, если вы попеременно пользуетесь скриптом `service` и командой `initctl`.



Возможно, вы обратили внимание на упоминание символических ссылок при описании работы с демонами в Ubuntu 14.04. Символические ссылки (symbolic links) – это указатели на файл, которые позволяют обращаться к одному и тому же файлу с помощью нескольких альтернативных имен (то есть используя различные имена в разных каталогах и даже в одном каталоге). При этом файл в действительности существует в единственном экземпляре на диске. Символические ссылки не являются характерной особенностью Ubuntu, они широко используются практически во всех дистрибутивах Linux. Systemd также использует символические ссылки при работе со своими юнитами.

Tlgm: @it_boooks

Далее рассмотрим работу с фоновыми сервисами в дистрибутиве CentOS 7.1.

Работа с фоновыми сервисами в CentOS 7.1

В CentOS 7.1 применяется systemd в качестве системы инициализации, поэтому работа с демонами во многом похожа на аналогичную работу в Debian GNU/Linux 8.x. Фактически основные команды systemctl в точности те же самые, хотя можно заметить различия в именах юнитов при выполнении команды `systemctl list-units` в этих двух дистрибутивах Linux. Следует всегда помнить об этих различиях при одновременном использовании CentOS 7.x и Debian 8.x в эксплуатационной производственной среде.

Одно из различий между Debian и CentOS состоит в том, что в CentOS имеется скрипт-обертка с именем `service`, который позволяет запускать, останавливать, перезапускать и проверять состояние демонов. Вероятно, этот скрипт-обертка (такая характеристика присвоена ему, потому что он не выполняет какой-либо реальной работы, а просто служит оберткой (wrapper) для вызова команды `systemctl`) включен в дистрибутив для обеспечения обратной совместимости, так как в предыдущих версиях CentOS система инициализации systemd не использовалась, а функции инициализации возлагались на команду с именем `service`. Следует отметить, что, несмотря на совпадение имен этого скрипта и команды `service` из дистрибутива Ubuntu, это два совершенно разных скрипта, их невозможно переносить между дистрибутивами из-за несовместимости.

Синтаксис этой команды: `service` *`service subcommand`*. Как и в Ubuntu, синтаксис скрипта в CentOS противоположен синтаксису команды `systemctl` *`subcommand service`*.

Прежде чем завершить раздел, посвященный работе с демонами, и перейти к обсуждению поддержки сетей в ОС Linux, приведем краткие описания нескольких команд, которые могут оказаться полезными.

Прочие команды для работы с демонами

Раздел, описывающий работу с демонами, мы завершаем кратким обзором нескольких полезных команд. Для получения полной информации обо всех ключах и параметрах этих команд обратитесь к страницам руководства (введите `man` *`имя-команды`* в командной строке). Вот эти команды:

- для вывода установленных сетевых соединений с демоном можно воспользоваться командой `ss`. Прежде всего она полезна тем, что показывает активные (слушающие – listening) сетевые сокеты, то есть предоставляет возможность убедиться в том, что сетевая конфигурация для конкретного демона (фонового сервиса) работает правильно. Используйте команду `ss -lnt` для вывода активных TCP-сокетов, а команду `ss -lnu` для вывода активных UDP-сокетов;
- команда `ps` используется для получения информации о текущих выполняемых процессах в системе.

Прежде чем перейти к следующему разделу, сделаем краткий обзор материала, рассмотренного выше:

Tlgm: @it_boooks

- предпосылки и история создания Linux;
- основы перемещения по файловой системе и концепция путей;
- основные операции с файлами (создание, копирование и перемещение, удаление файлов и каталогов);
- работа с фоновыми сервисами (демонами).

Следующая важная тема – работа с сетями в ОС Linux, при рассмотрении которой мы будем активно использовать знания, полученные при изучении предыдущей части главы.

Работа с сетями в ОС Linux

Ранее в этой главе отмечалось, что излагаемая здесь информация об операционной системе Linux прежде всего предназначена для быстрого освоения основ работы с Linux в контексте сетевой программируемости и автоматизации сети. Вероятнее всего, в Linux вы будете использовать такие инструменты, как Python, Ansible или Jinja (рассматриваемые в главах 4, 9 и 6 соответственно), а ваша Linux-система будет обмениваться данными по сети с разнообразными сетевыми устройствами. Поэтому описание работы с операционной системой Linux не может считаться полным без рассмотрения обеспечения работы с сетями в этой ОС. Ведь главной темой данной книги все-таки являются сети.

Работа с интерфейсами

Основным компонентом сетевой среды в ОС Linux является интерфейс (interface). Linux поддерживает несколько различных типов интерфейсов, но чаще всего это физические интерфейсы, VLAN-интерфейсы и интерфейсы мостов (шлюзов). Как и в большинстве случаев в Linux, конфигурирование различных типов интерфейсов осуществляется с помощью утилит, выполняемых в командной оболочке bash, то есть запускаемых из командной строки, а также посредством редактирования конкретных файлов конфигурации, содержащих обычный текст. Чтобы сделать изменения в конфигурации интерфейса постоянными, обычно требуется внести эти изменения в файл конфигурации. Сначала рассмотрим использование утилит командной строки, затем обсудим внесение постоянных изменений в файлы конфигурации интерфейсов.

Изменение конфигурации интерфейса из командной строки

Подобно тому, как многие дистрибутивы Linux стали использовать systemd в качестве основной системы инициализации, большинство дистрибутивов перешло на единый набор утилит командной строки для работы с сетевыми интерфейсами. Эти команды являются частью комплекта утилит `iproute2` (в CentOS пакет называется `iproute`, в Debian 8.1 и в Ubuntu 14.04 пакет называется `iproute2`). Этот набор утилит использует команду `ip` для замены функциональных возможностей ранее применяемых (в настоящее время объявленных устаревшими) команд, таких как `ifconfig` и `route` (в дистрибутивы Ubuntu 14.04 и CentOS 7.1 включены эти старые команды, но в дистрибутиве Debian 8.1 их уже нет).

Tlgm: @it_boooks

Более подробная информация о комплекте iproute2

Если вам необходима более подробная информация о комплекте утилит iproute2, то обратитесь к странице Википедии (https://en.wikipedia.org/wiki/Iproute2; https://ru.wikipedia.org/wiki/Iproute2).

Для конфигурирования интерфейсов предназначены две подкоманды утилиты `ip`: `ip link` – для просмотра и настройки состояния канала связи (соединения) с интерфейсом и `ip addr` – для просмотра и настройки конфигурации IP-адреса интерфейса. (Ниже в этом разделе мы рассмотрим некоторые другие формы команды `ip`.)

Рассмотрим несколько примеров, выполняющих конкретные задачи с использованием команд `ip` для конфигурирования интерфейса.

Вывод списка интерфейсов Можно воспользоваться либо командой `ip link`, либо командой `ip addr` для вывода всех интерфейсов в системе, хотя каждая команда выводит информацию, немного отличающуюся от вывода другой команды.

Если необходим список интерфейсов с указанием их состояния, следует использовать команду `ip link list`, как показано ниже:

```
[vagrant@centos ~]$ ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:17 brd ff:ff:ff:ff:ff:ff
3: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

По умолчанию «действием», если можно так выразиться, большинства (если не всех) команд `ip` является вывод списка элементов, с которыми вы работаете. Таким образом, если вам нужен список всех интерфейсов, то можно просто дать команду `ip link` вместо `ip link list`, а если требуется список всех маршрутов, то выполняется команда `ip route` вместо `ip route list`. В примерах приводятся полные версии команд в учебных целях для полной ясности.

По промпту в примере можно понять, что вывод команды получен в системе CentOS 7.1. Синтаксис команд одинаков во всех трех дистрибутивах, расматриваемых в данной главе, а выводимая ими информация почти одинакова (за исключением имен интерфейсов).

Обратите внимание на то, что в этом выводе показаны текущий список интерфейсов (как уже было сказано выше, CentOS присваивает интерфейсам имена, отличающиеся от имен интерфейсов в Debian и Ubuntu), текущие значения максимального размера блока передачи данных MTU (maximum transmission unit), текущее административное состояние (UP), MAC-адрес или физический адрес, а также некоторая другая информация.

Tlgm: @it_boooks

Вывод этой команды также сообщает текущее состояние интерфейса (обратите внимание на информацию в угловых скобках, следующую непосредственно за именем интерфейса):

- UP – интерфейс активен и доступен (enabled);
- LOWER_UP – канал связи (соединение) с интерфейсом активен;
- NO_CARRIER (в примере не показан) – интерфейс активен и доступен, но отсутствует соединение (нет канала связи).

Если у вас есть опыт работы с сетевым оборудованием, то вам, вероятно, известно различие между состояниями интерфейса «down» (отключен; неактивен) и «administratively down» (административно отключен). Если интерфейс неактивен, потому что нет соединения (канала связи), то после имени интерфейса в угловых скобках выводится характеристика состояния NO_CARRIER. Если интерфейс отключен административно (решением администратора), то вы не увидите характеристик UP, LOWER_UP, NO_CARRIER, а состояние будет определено как DOWN. В следующем разделе будет показано, как использовать команду ip link для административного отключения (запрещения) интерфейса (то есть перевода его в состояние administratively down).

Список интерфейсов можно также получить с помощью команды ip addr list, как показано ниже (этот вывод получен в системе Ubuntu 14.04 LTS):

```
vagrant@trusty:~$ ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0c:29:33:99:f6 brd ff:ff:ff:ff:ff:ff
    inet 192.168.70.205/24 brd 192.168.70.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe33:99f6/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0c:29:33:99:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.11/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe33:9900/64 scope link
       valid_lft forever preferred_lft forever
vagrant@trusty:~$
```

Мы видим, что команда ip addr list тоже выводит список интерфейсов в системе, а еще некоторую информацию о состоянии соединения (канала связи) и Ipv4/Ipv6-адреса, присвоенные конкретным интерфейсам.

И в команде ip link list, и в команде ip addr list есть возможность ограничить список одним конкретным интерфейсом, если добавить в команду его имя.

Tlgm: @it_boooks

Тогда команда принимает вид `ip link list` *interface* или `ip addr list` *interface*, как показано ниже:

```
vagrant@jessie:~$ ip link list eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:bf:af:1a brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

Вывод списка интерфейсов очень удобен, но с практической точки зрения еще более полезной является предоставляемая возможность изменения конфигурации интерфейса. В следующем разделе показано, как можно активизировать и отключить интерфейс.

Активизация/отключение интерфейса В дополнение к функции вывода интерфейсов команда `ip` предоставляет функцию управления состоянием интерфейса. Например, чтобы отключить (запретить) интерфейс, необходимо установить для него состояние `down` с помощью команды `ip link set`, как показано ниже:

```
[vagrant@centos ~]$ ip link set ens33 down
[vagrant@centos ~]$ ip link list ens33
3: ens33: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT
    qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

Обратите внимание на характеристику `state DOWN` и отсутствие характеристики `NO_CARRIER`. Это говорит о том, что рассматриваемый интерфейс административно отключен (запрещен), а не потерял работоспособность из-за критического сбоя (или отсутствия) соединения (канала связи). (В приведенном примере часть вывода `state DOWN` умышленно выделена полужирным шрифтом, чтобы ее было легче заметить.)

Для активизации (или повторной активизации) интерфейса ens33 нужно просто выполнить команду `ip link set` еще раз, но теперь задать состояние `up`:

```
[vagrant@centos ~]$ ip link set ens33 up
[vagrant@centos ~]$ ip link list ens33
3: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT qlen 1000
    link/ether 00:0c:29:d7:28:21 brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$
```

Установка значения MTU для интерфейса Если необходимо установить значение MTU для какого-либо интерфейса, то для этого снова потребуется команда `ip link` с подкомандой `set`. Полный синтаксис: `ip link set mtu` *MTU interface*.

Предположим, что необходимо обеспечить передачу jumbo-фреймов (с размером более 1500 байтов) для интерфейса ens33 в системе CentOS 7.x. Для этого выполняется следующая команда:

```
[vagrant@centos ~]$ ip link set mtu 9000 ens33
[vagrant@centos ~]$
```

Tlgm: @it_boooks

Как и в случаях применения других, ранее рассмотренных команд, это изменение происходит немедленно, но не является постоянным. Чтобы сделать изменение постоянным, необходимо внести его в файл конфигурации соответствующего интерфейса. Эта тема будет рассматриваться немного позже в разделе «Конфигурирование интерфейсов с помощью файлов конфигурации».

Присваивание IP-адреса интерфейсу Для присваивания (или удаления) IP-адреса интерфейсу используется команда `ip addr`. Мы уже пользовались этой командой для вывода списка интерфейсов и соответствующих им IP-адресов, а сейчас рассмотрим ее применение для добавления и удаления IP-адреса.

Для присваивания (добавления) IP-адреса интерфейсу предназначена команда `ip addr add` *`address`* `dev` *`interface`*. Например, если в системе Debian 8.1 необходимо присвоить (добавить) адрес 172.31.254.100/24 интерфейсу eth1, то выполняется следующая команда:

```
vagrant@jessie:~$ ip addr add 172.31.254.100/24 dev eth1
vagrant@jessie:~$
```

Если интерфейсу ранее уже был присвоен IP-адрес, то команда `ip addr add` просто добавляет новый адрес, не изменяя существующий. Поэтому в нашем примере, если интерфейсу eth1 уже был предварительно назначен адрес 192.168.100.10/24, выполнение предыдущей команды в такой конфигурации привело бы к следующему результату:

```
vagrant@jessie:~$ ip addr list eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0c:29:bf:af:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.10/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet 172.31.254.100/24 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:af24/64 scope link
       valid_lft forever preferred_lft forever
vagrant@jessie:~$
```

Для удаления IP-адреса из интерфейса применяется команда `ip addr del` *`address`* `dev` *`interface`*. В следующем примере удаляется адрес 172.31.254.100/24, который мы ранее назначили интерфейсу eth1:

```
vagrant@jessie:~$ ip addr del 172.31.254.100/24 dev eth1
vagrant@jessie:~$ ip addr list eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0c:29:bf:af:24 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.10/24 brd 192.168.100.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:af24/64 scope link
       valid_lft forever preferred_lft forever
vagrant@jessie:~$
```

Tlgm: @it_boooks

Как и для команды `ip link`, синтаксис команд `ip addr add` и `ip addr del` одинаков для всех трех дистрибутивов Linux, рассматриваемых в этой главе. Их вывод также идентичен в целом, хотя возможны различия в именах интерфейсов.

Мы продемонстрировали, как можно использовать команды ip для изменения конфигурации интерфейса. Если вы знакомы с процедурами конфигурирования сетевых устройств (если вы читаете эту книгу, то, вероятнее всего, знакомы), то можете рассматривать это как аналогию с динамическим изменением текущей конфигурации сетевого устройства. Но мы должны уметь сделать эти изменения конфигурации постоянными. Другими словами, мы пока еще не изменили изначальную конфигурацию. Поэтому в следующем разделе рассматривается, как можно вносить изменения в конфигурационные файлы интерфейсов в ОС Linux.

Конфигурирование интерфейсов с помощью файлов конфигурации

Чтобы изменения в конфигурации интерфейса стали постоянными и применялись при каждой загрузке системы, недостаточно использования одной лишь команды `ip`. Необходимо редактирование файлов конфигурации интерфейса, используемых при загрузке Linux для автоматической настройки и выполнения некоторых действий. Команды `ip` в основном одинаковы в различных дистрибутивах Linux, чего нельзя сказать, к сожалению, о файлах конфигурации сетевых интерфейсов – они могут отличаться, иногда весьма существенно.

Например, в группе RHEL/CentOS/Fedora и производных от них дистрибутивах компоненты конфигурации интерфейсов размещены в отдельных файлах, расположенных в каталоге */etc/sysconig/network-scripts*. Файлы конфигурации именуются по схеме `ifcfg-`*`interface`*, то есть имя интерфейса (например, eth0 или ens32) включено в имя файла. Содержимое файла конфигурации может выглядеть приблизительно следующим образом (пример взят из CentOS 7.1):

```
NAME="ens33"
DEVICE="ens33"
ONBOOT=yes
NETBOOT=yes
IPV6INIT=yes
BOOTPROTO=dhcp
TYPE=Ethernet
```

Ниже кратко описаны некоторые наиболее часто используемые директивы для файлов конфигурации сетевых интерфейсов в системах RHEL/CentOS/Fedora:

- `NAME` – удобное для чтения имя, которое видит пользователь, обычно используемое только в графических пользовательских интерфейсах (GUI) (это имя не отображается в выводе команд `ip`);
- `DEVICE` – имя физического устройства, для которого определяется данная конфигурация;
- `IPADDR` – IP-адрес, присваиваемый этому интерфейсу (если не используется протокол DHCP или BootP);

Tlgm: @it_boooks

- PREFIX – если IP-адрес присваивается статически, то этот параметр определяет префикс сети, используемый вместе с присваиваемым IP-адресом. (Вместо него можно использовать параметр NETMASK, но рекомендуется отдавать предпочтение PREFIX);
- BOOTPROTO – эта директива определяет, каким образом интерфейсу будет назначаться IP-адрес. Значение dhcp, показанное в примере выше, означает, что адрес будет назначен динамически по протоколу Dynamic Host Configuration Protocol (DHCP). Часто используется другое значение none, говорящий, что IP-адрес статически определяется в файле конфигурации интерфейса;
- ONBOOT – если задано значение yes, то интерфейс будет активизирован во время загрузки системы. Соответственно, значение no не позволяет активизировать интерфейс во время загрузки;
- MTU – определяет значение максимального размера передаваемого блока MTU для этого интерфейса;
- GATEWAY – определяет шлюз, используемый для этого интерфейса.

Существуют многие другие параметры конфигурации, но здесь описаны те, которые используются наиболее часто. Более подробно о параметрах конфигурации можно узнать из документа */usr/share/doc/initscripts-<version>/sysconig.txt* в установленной системе CentOS.

С другой стороны, в Debian и производных дистрибутивах, таких как Ubuntu, конфигурация интерфейса определяется в файле */etc/network/interfaces*. Ниже приведен пример файла конфигурации сетевого интерфейса из системы Ubuntu 14.04 LTS (использована команда cat для вывода всего содержимого этого файла на экран):

```
vagrant@trusty:~$ cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet static
      address 192.168.100.11
      netmask 255.255.255.0
vagrant@trusty:~$
```

Отметим, что в Debian и Ubuntu используется единственный файл для конфигурации всех сетевых интерфейсов. Каждый интерфейс размещен в отдельном «параграфе» (stanza) конфигурации, начинающемся со строки `auto` *`interface`*. В каждом параграфе конфигурации чаще всего используются следующие параметры и директивы (чтобы узнать обо всех параметрах и директивах кон-

Tlgm: @it_boooks

фигурации интерфейсов в системах Debian и Ubuntu, обратитесь к странице руководства `man 5 interfaces`):

- выбор метода назначения IP-адреса: для присваивания IP-адресов интерфейсам обычно используются директивы `inet dhcp` или `inet static`. В примере, показанном выше, для интерфейса eth0 определено использование протокола DHCP, в то время как интерфейсу eth1 IP-адрес назначается статически;
- параметр `netmask` определяет маску сети для присваиваемого IP-адреса (когда адрес назначается статически директивой `inet static`). Можно также воспользоваться форматом префикса (например, `192.168.100.10/24`) непосредственно в присваиваемом IP-адресе, тогда параметр netmask не нужен;
- директива `gateway` в конкретном параграфе конфигурации назначает шлюз по умолчанию в том случае, когда IP-адрес присваивается статически (с помощью `inet static`).

Если вы предпочитаете использовать отдельные файлы для хранения конфигурации интерфейсов, то существует также возможность разделения на файлы конфигурации, отдельно предназначенные для каждого интерфейса, подобно тому, как это сделано в системах RHEL/CentOS. В этом случае в файл */etc/network/interfaces* включается следующая строка:

```
source /etc/network/interfaces.d/*
```

Эта строка сообщает операционной системе, что файлы конфигурации для каждого отдельного интерфейса нужно искать в каталоге */etc/network/interfaces.d/* и обрабатывать их содержимое так, как если бы оно было напрямую включено в основной файл конфигурации сети. Файл */etc/network/interfaces* в Debian содержит такую строку по умолчанию (но указанный в ней каталог пуст, а конфигурация интерфейсов размещена в основном файле */etc/network/interfaces*). Благодаря наличию этой строки в основном файле можно без затруднений разместить конфигурационные параметры в отдельных файлах для каждого интерфейса.



Практическое использование отдельных файлов конфигурации для каждого интерфейса

Отдельные файлы конфигурации для каждого интерфейса могут предоставить дополнительную гибкость при использовании инструментов управления конфигурацией, таких как Chef, Puppet, Ansible или Salt. Это важные «орудия труда» для управления любыми системами, в том числе и Linux. При использовании этих инструментов возможно упрощение генерации отдельных файлов конфигурации для интерфейсов вместо попыток управления несколькими параграфами конфигурации в одном файле. Использование этих инструментальных средств будет более подробно рассматриваться в главе 9.

При внесении изменений в файл конфигурации сетевого интерфейса эти изменения не вступают в силу немедленно. (Если необходимы срочные изменения, то воспользуйтесь командой ip, описанной в предыдущем разделе, в до-

Tlgm: @it_boooks

полнение к внесению изменений в файлы конфигурации.) Чтобы изменения начали действовать, необходимо перезапустить сетевой интерфейс.

В Ubuntu 14.04 для этого используется команда `initctl`, описанная выше в разделе «Работа с демонами»:

```
vagrant@trusty:~$ initctl restart network-interface INTERFACE=interface
```

В CentOS 7.1 для той же цели применяется команда `systemctl`:

```
[vagrant@centos ~]$ systemctl restart network
```

В Debian 8.1 перезапуск сетевых интерфейсов выполняется похожей командой:

```
vagrant@jessie:~$ systemctl restart networking
```

Отметим, что в дистрибутивах, использующих systemd (CentOS и Debian 8.x), нет способа перезапуска одного конкретного интерфейса.

После перезапуска интерфейса изменения в конфигурации применяются и начинают действовать (это можно проверить с помощью соответствующих команд `ip`).

До сих пор мы рассматривали только физические интерфейсы, подобные eth0 или ens32. Но подобно тому, как операционная система Linux интерпретирует практически все сущности как файлы, подсистема поддержки сетевой среды Linux также интерпретирует многие сущности как интерфейсы. Одним из примеров может служить способ взаимодействия ОС Linux с виртуальными локальными сетями VLAN. Эту тему мы подробно рассмотрим в следующем разделе.

Использование интерфейсов VLAN

В одном из предыдущих разделов («Работа с демонами») было отмечено, что интерфейс является основным структурным компонентом в сетевой среде ОС Linux. В этом разделе мы рассмотрим интерфейсы VLAN, представляющие собой логические интерфейсы, которые позволяют одной Linux-системе одновременно обмениваться данными с несколькими виртуальными локальными сетями VLAN (virtual local area network) без необходимости предоставления отдельного физического интерфейса для каждой VLAN. Вместо этого Linux использует концепцию логических VLAN-интерфейсов, которые связываются и с физическим интерфейсом, и с соответствующим идентификатором по стандарту 802.1Q VLAN ID.

Вероятнее всего, вы уже знакомы с общей концепцией VLAN, поэтому здесь мы не будем рассматривать ее во всех подробностях. Если вам необходимо подробное справочное руководство по VLAN (и по многим другим сетевым концепциям), рекомендуем изучить книгу Packet Guide to Routing and Switching Брюса Хартпенса (Bruce Hartpence), изданную O'Reilly.

Создание, конфигурирование и удаление VLAN-интерфейсов Для создания VLAN-интерфейса используется команда `ip link add link` *`parent-device vlan-`*

Tlgm: @it_boooks

device type vlan id *vlan-id*. Очевидно, что это просто расширенная версия команды ip link, которая рассматривалась в нескольких предыдущих разделах текущей главы.

Но в этой расширенной команде имеются элементы, которые раньше не встречались, поэтому рассмотрим каждый из этих элементов более внимательно:

- *parent-device* – физический адаптер, с которым связывается логический VLAN-интерфейс. Это может быть что-то, подобное eth1 в системах Debian и Ubuntu или ens33 в системе RHEL/CentOS/Fedora;
- *vlan-device* – имя, присваиваемое логическому VLAN-интерфейсу. Существует общепринятое соглашение об именовании VLAN-интерфейсов: сначала записывается имя родительского устройства (parent-device), потом точка, за которой следует идентификатор VLAN ID. VLAN-интерфейсу, связанному с устройством eth1 и имеющему идентификатор VLAN ID 100, соответствует имя eth1.100;
- *vlan-id* – значение идентификатора по стандарту 802.1Q VLAN ID, назначаемое для данного логического интерфейса.

Рассмотрим пример. Предположим, что необходимо создать логический VLAN-интерфейс в системе Debian 8.x. Этот логический интерфейс будет связан с физическим интерфейсом eth2 и должен использовать идентификатор 802.1Q VLAN ID 150. Соответствующая команда выглядит следующим образом:

```
vagrant@jessie:~$ ip link add link eth2 eth2.150 type vlan id 150
vagrant@jessie:~$
```

После этого проверить существование логического VLAN-интерфейса можно с помощью команды ip link list (отметим, что в примере указано имя интерфейса как eth2.150@eth2, но при работе с этим интерфейсом необходимо использовать только часть перед символом @):

```
vagrant@jessie:~$ ip link list eth2.150
7: eth2.150@eth2: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN
    mode DEFAULT group default
    link/ether 00:0c:29:5f:d2:15 brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

Чтобы проверить (независимо от имени), действительно ли это логический VLAN-интерфейс, в команду ip link list следует добавить ключ -d, как показано ниже:

```
vagrant@jessie:~$ ip -d link list eth2.150
7: eth2.150@eth2: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN
    mode DEFAULT group default
    link/ether 00:0c:29:5f:d2:15 brd ff:ff:ff:ff:ff:ff
    vlan protocol 802.1Q id 150 <REORDER_HDR>
vagrant@jessie:~$
```

Для того чтобы этот VLAN-интерфейс стал действительно полнофункциональным, необходимо его активизировать (разрешить его использование) и присвоить ему IP-адрес:

Tlgm: @it_boooks


```
vagrant@jessie:~$ ip link set eth2.150 up
vagrant@jessie:~$ ip addr add 192.168.150.10/24 dev eth2.150
vagrant@jessie:~$
```

Разумеется, также необходимо создать соответствующую конфигурацию для физических коммутаторов, с которыми соединена данная система. В частности, порт коммутатора обязательно должен быть сконфигурирован как основной магистральный канал (trunk) VLAN и ориентирован на передачу VLAN 150. Команды для выполнения этих операций могут быть различными и зависят от производителя и конкретной модели коммутатора.

Как и для физического интерфейса, для логического VLAN-интерфейса, который активизирован и имеет присвоенный IP-адрес, добавляется маршрут в таблицу маршрутизации хоста:

```
vagrant@jessie:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.243
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.10
192.168.150.0/24 dev eth2.150  proto kernel  scope link  src 192.168.150.10
vagrant@jessie:~$
```

При удалении VLAN-интерфейса рекомендуется сначала отключить интерфейс (запретить его использование, то есть установить для него состояние `down`), затем удалить его, как показано ниже:

```
vagrant@jessie:~$ ip link set eth2.150 down
vagrant@jessie:~$ ip link delete eth2.150
vagrant@jessie:~$
```

Как уже было отмечено выше в этой главе, команды ip изменяют текущую (действующую) конфигурацию, но эти изменения не являются постоянными – после перезагрузки системы все созданные и сконфигурированные (как в предыдущих примерах) VLAN-интерфейсы исчезнут. Чтобы изменения стали постоянными, необходимо отредактировать файлы конфигурации интерфейсов.

В системе Debian/Ubuntu нужно просто добавить параграф в файл */etc/network/interfaces* или создать отдельный файл конфигурации в каталоге */etc/network/interfaces.d* (а также убедиться, что на этот файл есть ссылка в строке `source` в файле */etc/network/interfaces*). Параграф в общем файле конфигурации должен выглядеть приблизительно следующим образом:

```
auto eth2.150
iface eth2.150 inet static
  address 192.168.150.10/24
```

В системах RHEL/Fedora/CentOS необходимо в каталоге */etc/sysconig/network-scripts* создать файлы конфигурации для каждого интерфейса с соответствующими именами – в нашем случае это будет имя *ifcfg-eth2.150*. Содержимое файла конфигурации должно быть приблизительно таким:

```
VLAN=yes
DEVICE=eth2.150
```

Tlgm: @it_boooks


```
BOOTPROTO=static
ONBOOT=yes
TYPE=Ethernet
IPADDR=192.168.150.10
NETMASK=255.255.255.0
```

Примеры использования VLAN-интерфейсов VLAN-интерфейсы оказываются чрезвычайно удобными и полезными при использовании на Linux-хосте, который должен одновременно обмениваться данными с несколькими VLAN, и при этом желательно свести к минимуму требуемое количество портов коммутатора и физических интерфейсов. Например, если Linux-хост должен обмениваться данными в одной VLAN для доступа к веб-серверам, а в другой VLAN для доступа к серверам баз данных, то использование одного физического интерфейса с двумя логическими VLAN-интерфейсами является наилучшим решением (предполагается, что физический интерфейс обеспечивает достаточную пропускную способность).

В приложении А рассматриваются некоторые дополнительные примеры практического применения VLAN-интерфейсов.

Конфигурирование и управление таблицами IP-маршрутизации Linux-хоста – не менее важная задача, чем конфигурирование и управление сетевыми интерфейсами. Поэтому в следующем разделе подробно рассматриваются задачи маршрутизации.

Маршрутизация для конечного хоста

В дополнение к конфигурированию сетевых интерфейсов на Linux-хосте рассмотрим способы управления маршрутизацией в Linux-системе. Конфигурация интерфейсов и маршрутизация взаимосвязаны вполне естественным образом, но иногда необходимо конфигурировать некоторые задачи IP-маршрутизации отдельно от конфигурирования интерфейсов. Однако сначала рассмотрим, каким образом конфигурации интерфейсов влияют на конфигурацию маршрутизации хоста.

Команда `ip route` является главным инструментом просмотра и редактирования таблицы маршрутизации Linux-хоста, тем не менее команды `ip link` и `ip addr` также могут повлиять на содержимое таблицы маршрутизации.

Если необходимо просто просмотреть текущую таблицу маршрутизации, то следует выполнить команду `ip route list`, как показано ниже:

```
vagrant@trusty:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.11
vagrant@trusty:~$
```

Вывод этой команды сообщает пользователю о следующих фактах:

- адрес шлюза по умолчанию 192.168.70.2. Устройство eth0 используется для обмена данными со всеми неизвестными сетями через шлюз, за-

Tlgm: @it_boooks

данный по умолчанию. (Напомним, что в предыдущем разделе была указана возможность определения этого адреса по протоколу DHCP или с помощью специальной директивы конфигурации, например GATEWAY в системах RHEL/CentOS/Fedora или gateway в системах Debian/Ubuntu);
- устройству eth0 присвоен IP-адрес 192.168.70.205 – это интерфейс, который будет использоваться для обмена данными с сетью 192.168.70.0/24;
- устройству eth1 присвоен IP-адрес 192.168.100.11/24 – это интерфейс, который будет использоваться для обмена данными с сетью 192.168.100.0/24.

Если отключить (запретить) устройство eth1 с помощью команды ip link set eth1 down, то таблица маршрутизации хоста изменится автоматически:

```
vagrant@trusty:~$ ip link set eth1 down
vagrant@trusty:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
vagrant@trusty:~$
```

После отключения устройства eth1 в системе больше нет маршрута в сеть 192.168.100.0/24, так как таблица маршрутизации обновилась автоматически. Это вполне ожидаемый результат, но приведенный пример должен был наглядно продемонстрировать непосредственное воздействие команд ip link и ip addr на таблицу маршрутизации хоста.

Для изменений, которые не производятся полностью автоматически, используется команда ip route. Что подразумевается под «не полностью автоматическими изменениями»? Приведем несколько вариантов из реальной практики:
- добавление статического маршрута в некоторую сеть через конкретный интерфейс;
- удаление статического маршрута в некоторую сеть;
- изменение шлюза, заданного по умолчанию.

А теперь рассмотрим несколько конкретных примеров для этих вариантов.

Предположим, что используется та же конфигурация, которая была показана в начале раздела, – интерфейсу eth0 присвоен Ipv4-адрес из сети 192.168.70.0/24, а интерфейс eth1 имеет Ipv4-адрес из сети 192.168.100.0/24. При такой конфигурации вывод команды ip route list выглядит следующим образом:

```
vagrant@trusty:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.205
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.11
vagrant@trusty:~$
```

Если изобразить эту конфигурацию в виде сетевой диаграммы, то она будет выглядеть, как показано на рис. 3.3.

Tlgm: @it_boooks

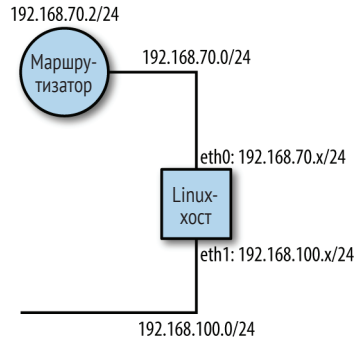


Рис. 3.3 ❖ Пример топологической схемы сети

Теперь предположим, что в сеть 192.168.100.0/24 добавляется новый маршрутизатор, а сеть, с которой рассматриваемый хост должен обмениваться данными (используя адрес подсети 192.168.101.0/24), расположена позади этого маршрутизатора. На рис. 3.4 показана новая топологическая схема сети.

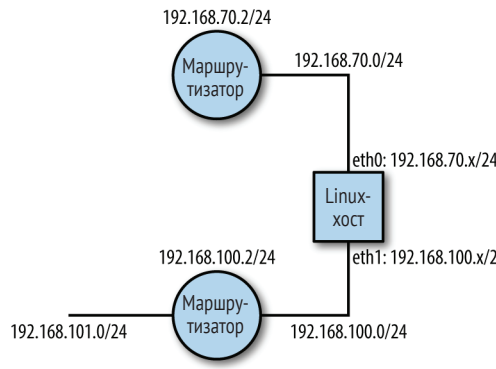


Рис. 3.4 ❖ Измененная топологическая схема сети

Существующая таблица маршрутизации рассматриваемого хоста не позволяет обмениваться данными с новой сетью, так как отсутствует маршрут к этой сети. Система Linux будет направлять трафик в шлюз по умолчанию, но шлюз не установил соединение с новой сетью. Чтобы устранить проблему, необходимо добавить маршрут к новой сети через интерфейс eth1 нашего хоста, как показано ниже:

```
vagrant@jessie:~$ ip route add 192.168.101.0/24 via 192.168.100.2 dev eth1
vagrant@jessie:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.204
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.10
```

Tlgm: @it_boooks


```
192.168.101.0/24 via 192.168.100.2 dev eth1
vagrant@jessie:~$
```

Обобщенная форма этой команды: `ip route add` *`destination-net`* `via` *`gateway-address`* `dev` *`interface`*.

Приведенная выше команда сообщает Linux-хосту (в нашем примере это система Debian), что он может обмениваться данными с сетью 192.168.101.0/24 через IP-адрес 192.168.100.2, присвоенный интерфейсу eth1. Теперь хост «знает» маршрут к новой сети через соответствующий маршрутизатор и может обмениваться данными с системами в этой сети. Если топологическая схема сети снова изменится, например будут подключены еще один маршрутизатор и новая сеть, как показано на рис. 3.5, то потребуется всего лишь добавить соответствующий маршрут в таблицу.

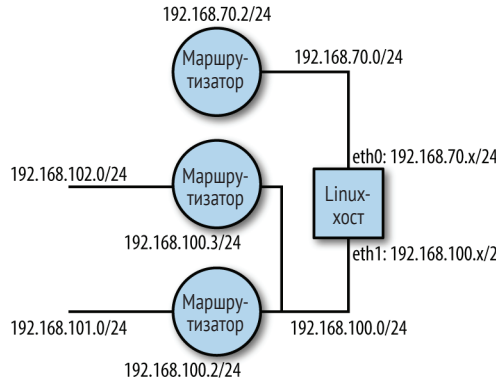


Рис. 3.5 ❖ Окончательная топологическая схема сети

Для соответствия окончательной топологической схеме сети необходимо выполнить следующую команду:

```
vagrant@jessie:~$ ip route add 192.168.102.0/24 via 192.168.100.3 dev eth1
vagrant@jessie:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.204
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.10
192.168.101.0/24 via 192.168.100.2 dev eth1
192.168.102.0/24 via 192.168.100.3 dev eth1
vagrant@jessie:~$
```

Чтобы сделать новые добавленные маршруты постоянными (напомним, что команды `ip`, как правило, не вносят постоянных изменений в конфигурацию), следует добавить все описанные выше команды в параграф конфигурационного файла */etc/network/interfaces* для устройства eth1, как показано ниже (в системах RHEL/Fedora/CentOS необходимо добавить эти команды в файл конфигурации */etc/sysconig/network-scripts/ifcfg-eth1*):

Tlgm: @it_boooks


```
auto eth1
iface eth1 inet static
      address 192.168.100.11
      netmask 255.255.255.0
  up ip route add 192.168.101.0/24 via 192.168.100.2 dev $IFACE
  up ip route add 192.168.102.0/24 via 192.168.100.3 dev $IFACE
```

В командах этого параграфа конфигурации переменная `$IFACE` ссылается на конкретный интерфейс, для которого определяется конфигурация, а директива `up` сообщает системам Debian/Ubuntu о необходимости выполнения этих команд после активизации (подключения) интерфейса. После добавления приведенных выше строк в файл конфигурации указанные маршруты будут автоматически добавляться в таблицу маршрутизации при каждом запуске системы.

Если по каким-либо причинам необходимо удалить маршруты из таблицы маршрутизации, то можно также воспользоваться командой `ip route`, но на этот раз с подкомандой `delete`:

```
[vagrant@centos ~]$ ip route del 192.168.103.0/24 via 192.168.100.3
```

Обобщенная форма команды удаления маршрута: `ip route del` *`destination-net`* `via` *`gateway-address`*.

Изменение шлюза, заданного по умолчанию, также можно выполнить с помощью команды `ip route`. (Но следует отметить, что можно изменить шлюз по умолчанию еще и посредством редактирования файлов конфигурации интерфейсов, при этом внесенные изменения становятся постоянными. Команда ip route дает немедленный эффект, но такое изменение не является постоянным.) Команда изменения шлюза, определяемого по умолчанию, должна иметь приблизительно следующий вид (предполагается, что в файле конфигурации уже имеется запись о шлюзе по умолчанию):

```
vagrant@trusty:~$ ip route del default via 192.168.70.2 dev eth0
vagrant@trusty:~$ ip route add default via 192.168.70.1 dev eth0
```

В этих командах ключевое слово `default` используется для ссылки на целевой адрес 0.0.0.0/0.

В ОС Linux также поддерживается так называемая стратегия маршрутизации (policy routing), предоставляющая возможность работы с несколькими таблицами маршрутизации в совокупности с правилами, определяющими выбор конкретной таблицы маршрутизации для текущего использования. Например, предположим, что для каждого интерфейса в системе предназначен отдельный шлюз, заданный по умолчанию. Используя механизм стратегии маршрутизации, можно сконфигурировать систему таким образом, что для eth0 будет использоваться одна таблица маршрутизации (следовательно, конкретно заданный по умолчанию шлюз), а для eth1 – другая таблица маршрутизации (соответственно, и другой шлюз по умолчанию). Стратегия маршрутизации представляет собой слишком обширную тему, поэтому здесь она не рассмат-

Tlgm: @it_boooks

ривается, но если вы хотите узнать подробнее, как она работает, то обратитесь к страницам руководства или к функциям вывода подсказки команд `ip rule` и `ip route` (то есть выполните команды `man ip rule` и `man ip route`).

В этом разделе рассматривалась IP-маршрутизация для конечного сетевого хоста, но Linux-система может использоваться как полнофункциональный IP-маршрутизатор. Как и стратегия маршрутизации, эта тема весьма обширна, тем не менее мы рассмотрим основные элементы конфигурирования Linux-системы как маршрутизатора в следующем разделе.

Конфигурация маршрутизатора

Почти во всех современных дистрибутивах Linux по умолчанию отключена (запрещена) функция IP forwarding (IP-переадресация), поскольку большинству пользователей эта функция не нужна. Тем не менее ОС Linux может выполнять IP-переадресацию (IP forwarding), то есть может работать как маршрутизатор (router), объединяющий несколько IP-подсетей и передающий (перенаправляющий) трафик между несколькими подсетями. Чтобы воспользоваться этой функциональной возможностью, сначала необходимо разрешить (активизировать) функцию IP-переадресации.

Для проверки состояния функции IP-переадресации (запрещена или разрешена) необходимо выполнить следующую команду (она работает в Debian, Ubuntu и CentOS, но в других системах соответствующий файл может размещаться в различных каталогах, поэтому полный путь может отличаться от указанного здесь):

```
vagrant@trusty:~$ /sbin/sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
vagrant@trusty:~$ /sbin/sysctl net.ipv6.conf.all.forwarding
net.ipv6.conf.all.forwarding = 0
vagrant@trusty:~$
```



В тех случаях, когда в альтернативных дистрибутивах Linux выполняемый файл указанной выше команды размещается в другом каталоге, найти его поможет команда `which`, уже упомянутая ранее в этой главе (при условии что этот файл находится в одном из каталогов, включенных в путь поиска).

В обоих случаях при выполнении команды выводится значение `0`, означающее, что функция IP-перенаправления запрещена. Разрешить (подключить) эту функцию можно без перезагрузки системы, используя следующую команду (но это разрешение будет непостоянным – функция снова будет отключена после перезагрузки системы):

```
[vagrant@centos ~]$ systctl -w net.ipv4.ip_forward=1
```

Эта команда похожа на команду `ip`, рассматриваемую выше, в том, что при ее выполнении изменение вступает в силу немедленно, но измененное состояние не сохраняется после перезагрузки Linux-системы. Чтобы это изменение стало постоянным, необходимо отредактировать файл */etc/sysctl.conf* или по-

Tlgm: @it_boooks

местить соответствующий файл конфигурации в каталог */etc/sysctl.d*. В любом случае, нужно добавить следующую строку в файл */etc/sysctl.conf* или в файл конфигурации в каталоге */etc/sysctl.d*:

```
net.ipv4.ip_forward = 1
```

Чтобы разрешить использование функции IPv6-перенаправления, в файл конфигурации добавляется следующая строка:

```
net.ipv6.conf.all.forwarding = 1
```

Затем можно перезагрузить Linux-хост, чтобы изменение начало действовать, или выполнить команду `sysctl -p <путь к файлу с новым установленным значением>`.

Единственный файл конфигурации или несколько конфигурационных файлов

Выше несколько раз отмечалось, что иногда в дистрибутивах Linux можно использовать отдельные файлы конфигурации в специальном каталоге (например, */etc/network/interfaces.d* или */etc/sysctl.d*). Какой способ лучше? Иногда эту тему обсуждают системные администраторы Linux и перечисляют достоинства и недостатки каждого способа. Использование отдельных файлов конфигурации может дать больше преимуществ при применении какого-либо инструментального средства управления конфигурацией (как инструмент управления самими файлами и их содержимым), но и другой способ в этом случае вполне приемлем.

После активизации функции IP-перенаправления Linux-система начинает работать как маршрутизатор. Но при этом она способна выполнять только статическую маршрутизацию, поэтому необходимо применять команду `ip route` для передачи всех необходимых инструкций и информации о маршрутизации, чтобы правильно регулировать трафик. Однако для Linux-систем существуют также демоны, реализующие протокол динамической маршрутизации и позволяющие Linux-маршрутизатору присоединиться к поддержке протоколов динамической маршрутизации, таким как BGP или OSPF. Наиболее распространенными средствами поддержки сред динамической маршрутизации в Linux являются Quagga (http://www.nongnu.org/quagga) и BIRD (http://bird.network.cz).

Используя такие средства, как механизм IPTables (или его преемник NFTables (http://netfilter.org/projects/nftables/)), можно добавить в систему функциональные возможности, подобные NAT (Network Address Translation) и спискам управления доступом (ACL – access control lists).

В дополнение к возможности управления трафиком на уровне 3 в ОС Linux также имеется возможность коммутации трафика, то есть объединения нескольких Ethernet-сегментов на уровне 2. В следующем разделе рассматриваются основы функционирования механизма коммутации в Linux.

Tlgm: @it_boooks

Коммутация

В ОС Linux мост (bridge; однопотоковый коммутатор) обеспечивает возможность объединения нескольких сетевых сегментов независимым от какого-либо протокола способом, таким образом, мост работает на уровне 2 модели OSI, а не на более высоких уровнях. Режим бриджинга (bridging), особенно многопортового прозрачного бриджинга, широко используется в современных центрах данных в форме сетевых коммутаторов (switches), наиболее частым способом применения режима бриджинга в Linux-системе являются различные формы виртуализации (реализуемые через гипервизор KVM или другими средствами, например механизмом контейнеров Linux). Поэтому здесь мы лишь кратко рассмотрим основы бриджинга, и только в контексте виртуализации.

Практические варианты применения режима бриджинга

Прежде чем перейти к подробностям создания и конфигурирования мостов, рассмотрим практический пример использования моста в ОС Linux.

Предположим, что имеется Linux-хост с двумя физическими интерфейсами (в этом примере для физических интерфейсов используются имена eth0 и eth1). Сразу после создания моста (сам процесс создания будет описан в следующем разделе) Linux-хост выглядит так, как показано на рис. 3.6.

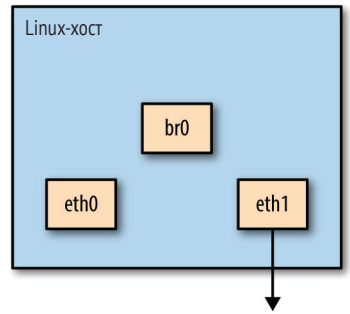


Рис. 3.6 ❖ Linux-мост без интерфейсов

Мост существует, но в действительности пока еще не может выполнять никаких действий. Напомним, что мост предназначен для объединения сетевых сегментов, поэтому без сегментов, присоединенных к мосту, невозможно что-либо сделать. Необходимо добавить интерфейсы, то есть связать их с мостом.

Допустим, что добавляется интерфейс eth1 к мосту br0. Полученная конфигурация показана на рис. 3.7.

Tlgm: @it_boooks

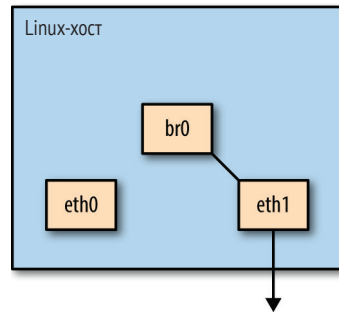


Рис. 3.7 ❖ Linux-мост
с одним физическим интерфейсом

Если бы сейчас мы подключили виртуальную машину (VM) к этому мосту (в приложении А описываются некоторые подробности использования моста вместе с виртуальной машиной; обычно это выполняется с помощью KVM (http://www.linux-kvm.org/page/Main_Page) и Libvirt (http://libvirt.org/)), то конфигурация приняла бы вид, показанный на рис. 3.8.

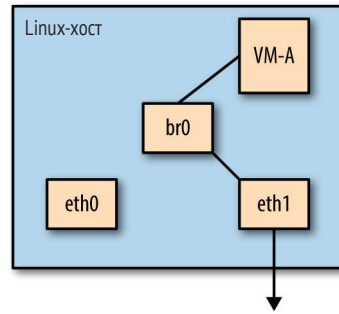


Рис. 3.8 ❖ Linux-мост с физическим интерфейсом
и виртуальной машиной

В этой окончательной конфигурации мост br0 соединяет (или обеспечивает режим бриджинга, если вы предпочитаете этот термин) сетевой сегмент для виртуальной машины и сетевой сегмент для физического интерфейса, обеспечивая создание единого широковещательного домена уровня 2 от виртуальной машины до сетевой карты (NIC) (и далее в физическую сеть). Предоставление сетевых соединений виртуальным машинам является весьма частым вариантом применения мостов Linux, но далеко не единственным. Также можно использовать Linux-мост для подключения к беспроводной сети (через беспроводной интерфейс на Linux-хосте), к сети Ethernet с соединением через обычную сетевую карту (NIC).

Tlgm: @it_boooks

После описания основных принципов применения мостов в ОС Linux можно перейти к рассмотрению процедур создания и конфигурирования мостов.

Создание и конфигурирование Linux-мостов

Для конфигурирования Linux-мостов используется та же утилита `ip`, которую мы ранее рассматривали в применении к конфигурированию и управлению сетевыми интерфейсами. Напомним, что в разделе «Работа с интерфейсами» мы определили интерфейсы как основные структурные компоненты сетевой среды Linux. Это определение остается справедливым и здесь, так как мосты интерпретируются операционной системой как один из типов интерфейсов.



Что такое brctl

Возможно, вы знакомы с более старыми командами, используемыми для работы с мостами, в частности с командой `brctl`. Почти так же, как команда `ip` заменила более старую команду `ifconfig`, команды `ip` и `bridge` (последняя будет рассматриваться ниже в этом разделе) заменили старую команду `brctl`. Необходимо отметить, что утилита `brctl` остается доступной в большинстве современных дистрибутивов Linux и может по-прежнему использоваться для работы с мостами. Но в этом разделе все внимание сосредоточено на новых командах из пакетов `iproute2`.

Для создания моста используется команда ip link с подкомандой add, как показано ниже:

```
vagrant@jessie:~$ ip link add name имя-моста type bridge
```

Эта команда создает мост, не содержащий интерфейсов (конфигурация, показанная на рис. 3.7 в предыдущем разделе). Проверка существования моста осуществляется командой `ip link list`. Например, если при создании моста было задано имя `br0`, то для проверки его состояния выполняется следующая команда:

```
vagrant@jessie:~$ ip link list br0
5: br0: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN mode DEFAULT
    group default
    link/ether 00:0c:29:5f:d2:15 brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

Обратите внимание на то, что для нового интерфейса моста указано состояние `DOWN`, поэтому необходимо воспользоваться командой `ip link set` *имя-моста* `up` для активизации этого моста.

После создания моста можно еще раз применить команду `ip link` для присоединения физического интерфейса к мосту. Обобщеннный синтаксис такой команды: `ip link set` *имя-интерфейса* `set master` *имя-моста*. Если необходимо подключить интерфейс `eth1` к мосту `br0`, то команда должна выглядеть следующим образом:

```
vagrant@jessie:~$ ip link set eth1 master br0
vagrant@jessie:~$
```

Tlgm: @it_boooks

После этого конфигурация выглядит так, как показано на рис. 3.7.

После присоединения физического интерфейса к мосту становится полезной команда `bridge` (часть того же пакета `iproute2`, который содержит утилиту `ip`). В этой главе уже отмечалось, что в различных дистрибутивах Linux некоторые команды размещаются в различных местах. В нашем случае в Debian и Ubuntu команда `bridge` находится в каталоге */sbin*, тогда как в CentOS она располагается в каталоге */usr/sbin*. Обычно это не вызывает никаких проблем, но в путь поиска по умолчанию для обычного пользователя в системе Debian 8.1 не включен каталог */sbin* (если только вы не работаете как суперпользователь root, что крайне не рекомендуется). Поэтому в Debian вам придется использовать полный путь (*/sbin/bridge*) или отредактировать свой путь поиска, чтобы включить в него каталог */sbin*. Далее предполагается, что вы отредактировали свой путь поиска, поэтому в последующих примерах не указывается полное путевое имя для утилиты `bridge`.

С помощью утилиты `bridge` можно просмотреть все интерфейсы, которые являются частью моста:

```
vagrant@trusty:~$ bridge link
3: eth1 state UP : <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br0 state
    forwarding priority 32 cost 4
vagrant@trusty:~$
```

Команда `bridge link` показывает все интерфейсы, присоединенные к мосту. Чтобы увидеть только один конкретный интерфейс, нужно выполнить команду `bridge link show dev` *имя-интерфейса*. Кроме того, команда `bridge` позволяет редактировать определенные свойства присоединенных интерфейсов, например разрешать/запрещать обработку Bridge Protocol Data Units (BPDU) или разрешать/запрещать возможность обратного перенаправления трафика из порта, на котором он был принят (hairpinning или NAT loopback). Более подробно о применении команды `bridge` можно узнать на соответствующей странице руководства (`man bridge`).

Для удаления интерфейса из моста также используется команда `ip link`, как показано ниже:

```
[vagrant@centos ~]$ ip link set interface-name nomaster
[vagrant@centos ~]$
```

Наконец, для удаления самого моста выполняется команда `ip link del` с указанием имени удаляемого моста. Если необходимо удалить мост `br0`, то команда выглядит следующим образом:

```
vagrant@trusty:~$ ip link del br0
vagrant@trusty:~$
```

Отметим, что перед удалением моста не обязательно удалять из него все интерфейсы.

Все описанные выше команды создают непостоянные конфигурации. Чтобы сделать эти конфигурации постоянными, необходимо вернуться к разделу

Tlgm: @it_boooks

«Изменение конфигурации интерфейса из командной строки» и вспомнить, что в ОС Linux мост интерпретируется как один из типов интерфейсов, только в данном случае это логический интерфейс, а не физический.

Поскольку ОС Linux интерпретирует мосты как интерфейсы, для их конфигурирования используются те же самые типы файлов, которые рассматривались ранее: в RHEL/CentOS/Fedora это файл в каталоге */etc/sysconfig/network-scripts*, а в Debian/Ubuntu это параграф конфигурационного файла */etc/network/interfaces* (или отдельный файл конфигурации в подкаталоге */etc/network/interfaces.d*). Рассмотрим, как должна выглядеть конфигурация моста в CentOS и в Debian (в Ubuntu конфигурация моста почти аналогична Debian).

В системе CentOS 7.1 необходимо создать в каталоге */etc/sysconfig/network-scripts* файл конфигурации для требуемого моста. Например, если нужно создать мост `br0`, то создается файл с именем *ifcfg-br0*. Ниже приведен пример содержимого файла конфигурации для моста как одного из типов интерфейсов:

```
DEVICE=br0
TYPE=Bridge
ONBOOT=yes
BOOTPROTO=none
IPV6INIT=no
IPV6_AUTOCONF=no
DELAY=5
STP=yes
```

При такой конфигурации создается мост с именем `br0`, для которого разрешено использование протокола STP (Spanning Tree Protocol – протокол остовного дерева). Для добавления интерфейсов в этот мост необходимо изменить файлы конфигурации интерфейсов, которые должны стать частью этого моста. Например, если частью моста `br0` должен стать интерфейс `ens33`, то файл конфигурации для интерфейса `ens33` может выглядеть так:

```
DEVICE=ens33
ONBOOT=yes
HOTPLUG=no
BOOTPROTO=none
TYPE=Ethernet
BRIDGE=br0
```

В этом файле конфигурации параметр `BRIDGE` связывает данный интерфейс `ens33` с мостом `br0`.



Протокол остовного дерева в ОС Linux

Возможно, вы заметили, что в предыдущем примере файла конфигурации для моста в CentOS 7 разрешено использование протокола STP. В ядро Linux включена реализация протокола STP, но эта реализация постепенно выводится из использования, а заменяют ее реализации в пространстве пользователя. Поэтому для управления устаревающей реализацией в пространстве ядра можно использовать только файлы конфигурации или старую утилиту `brctl`.

Tlgm: @it_boooks

Обратите внимание на то, что ни для моста `br0`, ни для интерфейса `ens33` не назначается IP-адрес. Наиболее вероятное объяснение этого факта: в обычном сетевом коммутаторе стандартный порт коммутации, сконфигурированный для уровня 2 (канального), не имеет IP-адреса. Поэтому воспроизведенная в примере конфигурация объясняется следующим образом: `br0` – это коммутатор, а `ens33` – порт только для уровня 2, являющийся частью коммутатора.

Если бы необходимо было все-таки присвоить IP-адрес (возможно, для осуществления управления или потому, что желательно обеспечить еще и функциональность уровня 3 (сетевого) в ОС Linux), то можно присвоить IP-адрес мосту, но не интерфейсам, являющимся частью моста. И снова можно воспользоваться аналогией с обычным сетевым оборудованием – для управления коммутатором назначается IP-адрес, но отдельные порты коммутации, работающие только на уровне 2, по-прежнему остаются неадресуемыми по протоколу IP. Чтобы присвоить IP-адрес интерфейсу моста, просто добавьте параметры `IPADDR`, `NETMASK` и `GATEWAY` в файл конфигурации соответствующего моста.

В системе Debian (соответственно, и в Ubuntu) все почти то же самое. Для создания и настройки моста в Debian обычно добавляется соответствующий параграф в файл конфигурации */etc/network/interfaces* для конфигурирования самого моста, как показано ниже:

```
iface br0 inet manual
  up ip link set $IFACE up
  down ip link set $IFACE down
  bridge-ports eth1
```

Здесь создается мост `br0` с присоединением к нему интерфейса `eth1`. Отметим, что для интерфейсов, являющихся частью моста, не требуются специальные отдельные параграфы в файле конфигурации.

Если необходимо присвоить IP-адрес интерфейсу моста, то нужно просто заменить часть первой директивы `inet manual` на `inet dhcp` (чтобы использовать протокол DHCP) или на `inet static` (для присваивания статического адреса). Если присваивается статический адрес, то также необходимо включить соответствующие строки конфигурации для присваивания IP-адреса (то есть директивы `address`, `netmask` и необязательную директиву `gateway`).

После завершения создания всех необходимых файлов конфигурации для моста Linux эта конфигурация всегда будет восстанавливаться при загрузке системы, то есть станет постоянной. Проверить состояние конфигурации можно с помощью команды `ip link list`.

Практические примеры и более подробную информацию об использовании мостов в ОС Linux можно найти в приложении А.

Tlgm: @it_boooks

Резюме

В этой главе была кратко описана история создания ОС Linux, а также обоснована важность понимания основ работы в Linux для дальнейшего изучения сетевой программируемости и автоматизации. Также была предоставлена основная информация о взаимодействии с ОС Linux, о работе с демонами Linux и о конфигурировании сетевой среды Linux. Рассматривалось использование Linux-системы в качестве маршрутизатора, а также обсуждались функциональные возможности моста Linux.

В начале главы была отмечена одна из причин, по которой авторы считают важным включение информации об ОС Linux в данную книгу: некоторые рассматриваемые здесь инструменты берут свое начало и наиболее эффективно применяются именно в этой операционной системе. В следующей главе мы рассмотрим один из таких инструментов – язык программирования Python.

Tlgm: @it_boooks

Глава 4

Изучение языка программирования Python для применения в сетевой среде

Для сетевого инженера сейчас наступает самое подходящее время для изучения автоматизации и освоения программирования (написания кода на одном из языков программирования). В главе 1 было особо подчеркнуто текущее фундаментальное изменение сетевой отрасли. С конца 1990-х гг. по 2010 г. эта отрасль почти не изменялась, ни архитектурно, ни с точки зрения выполнения сетевых операций. В этот период сетевые инженеры без тени сомнения вводили в командной строке сотни и даже тысячи команд конфигурирования сетевых устройств и устранения проблем в их работе. Но можно ли назвать такой подход разумным?

Постепенно приобретали смысл и становились удобными сетевые операции, которые способны считывать и записывать некоторый программный код. Вообще говоря, написание скриптов или нескольких строк программного кода для сбора информации в сети или для внесения изменений не являются каким-то новшеством. Такой подход применялся в течение многих лет. Сетевые инженеры, которые воспользовались этой возможностью, то есть написанием кода на выбранном языке программирования, обучались работе с неструктурированным текстом, используя сложные методики синтаксического разбора (парсинга), регулярные выражения и SNMP-запросы к базам данных MIB в скриптах. Если вы сами попытаетесь проделать такую работу, то вскоре обнаружите, что это вполне возможно, но применение регулярных выражений и методик синтаксического анализа текста является чрезвычайно утомительной задачей и требует больших затрат времени.

К счастью, ситуация развивалась в правильном направлении, и препятствия, мешающие внедрению автоматизации сети, становились не такими

Tlgm: @it_boooks

неустранимыми, какими казались раньше. Мы наблюдаем усовершенствования, вносимые производителями сетевого оборудования, а также появление инструментальных программных средств с открытым исходным кодом, доступных для использования в целях автоматизации сети. Многие такие усовершенствования и инструментальные средства рассматриваются в данной книге. Например, сейчас предлагаются прикладные программные интерфейсы (API) для сетевых устройств, библиотеки языка Python, поддерживаемые как производителями, так и независмыми сообществами, и бесплатные инструментальные средства с открытым исходным кодом, которые предоставляют любому сетевому инженеру доступ к развивающейся экосистеме и возможность начать процесс автоматизации сети. В конечном итоге это означает, что вам придется писать меньше кода, чем ранее, следовательно, ускоряется разработка и уменьшается количество ошибок.

Прежд чем начать изучение основ языка Python, необходимо попытаться найти ответ на важный вопрос, часто возникающий в дискуссиях сетевых инженеров: должны ли сетевые инженеры уметь писать программный код?

Должны ли сетевые инженеры уметь писать программный код?

К сожалению, авторы не могут дать определенный ответ (да или нет) на этот вопрос. Конечно, мы включили в книгу большую главу по языку Python и огромное количество примеров, демонстрирующих использование Python для обмена данными с сетевыми устройствами с помощью сетевых API и широко распространенных платформ DevOps, таких как Ansible, Salt и Puppet, но мы уверены в том, что изучение основ любого языка программирования весьма полезно. Кроме того, мы считаем, что умение писать программный код станет еще более полезным навыком, поскольку сетевая отрасль и вся IT-индустрия в целом продолжают стремительно изменяться. Поэтому Python кажется нам наиболее удачным выбором для начального обучения программированию.



Следует особо отметить, что мы не отдаем каких-либо особых предпочтений и не создаем «технокульта» языка Python. Но мы считаем, что для задач автоматизации сети выбор этого языка является наиболее удачным решением по нескольким причинам. Во-первых, Python – язык с динамической типизацией, позволяющий создавать и использовать объекты (такие как переменные и функции), где и когда это необходимо, то есть нет необходимости предварительно определять такие объекты до начала их реального использования. Это упрощает процесс обучения. Во-вторых, код на языке Python легко читается человеком. Весьма часто можно видеть условные выражения вида `if device in device_list:`, и такое выражение без труда позволяет понять, что здесь проверяется наличие устройства в заданном списке устройств. Кроме того, производители сетевого оборудования и участники проектов с открытым исходным кодом создают великолепный набор библиотек и инструментальных средств на языке Python. Все эти факты подтверждают преимущества изучения программирования на языке Python.

Tlgm: @it_boooks

Но в действительности обсуждаемый здесь вопрос должен быть поставлен так: должен ли каждый сетевой инженер знать, как прочитать и написать простой скрипт? На такой вопрос можно дать вполне определенный ответ: да. Должен ли каждый сетевой инженер становиться разработчиком программного обеспечения? Определенно нет. Для многих сетевых инженеров какие-то технические дисциплины более притягательны, чем прочие, поэтому некоторые сетевые инженеры могут переквалифицироваться в разработчиков ПО, но инженеры всех специальностей (не только сетевые) не должны избегать изучения исходного кода на языках, подобных Python или Ruby, или даже кода на более сложных языках, таких как C или Go. Системные администраторы уже достаточно хорошо знакомы с использованием в качестве инструмента, позволяющего им более эффективно выполнять свою работу, скриптов на языке командной оболочки bash и на языках Python, Ruby и PowerShell.

С другой стороны, умение использовать скрипты не является основной характеристикой сетевых администраторов (и это одна из главных причин написания данной книги). Вполне очевидно, что развивается и сетевая отрасль, и сами инженеры, поэтому каждый сетевой инженер должен быть в большей степени ориентирован на платформу DevOps, чтобы в итоге не оказаться в неопределенном «среднем» положении, – ни разработчик, ни сетевой инженер старой школы, применяющий только интерфейс командной строки. Вы можете освоить инструментальные средства управления конфигурации и автоматизации с открытым исходным кодом, а затем добавлять небольшие фрагменты кода по необходимости, чтобы управлять рабочими процессами и потоками и автоматизировать задачи в конкретной эксплуатационной среде.



Если в вашей организации установлены жесткие требования на основе размера, масштабируемости, совместимости или управления, то не следует (и не рекомендуется) увлекаться написанием специализированного программного обеспечения «для всего» и создавать собственную доморощенную платформу автоматизации. Это нерациональная трата времени. Рекомендуется изучить и хорошо понимать компоненты, используемые в программировании, в разработке программного обеспечения. Особенно тщательно следует изучить основы программирования, такие как базовые типы данных, которые являются общепринятыми во всех инструментах и языках. Именно такие основы рассматриваются в этой главе на примере конкретного языка программирования Python.

После того как вы полностью осознали, что сетевая отрасль быстро меняется, что устройства имеют API, можно переходить к началу обучения кодированию. В этой главе представлены основные структурные компоненты, необходимые для быстрого освоения программирования на языке Python.

В следующих разделах главы будут рассматриваться такие темы:

- использование интерактивного интерпретатора Python;
- изучение типов данных языка Python;
- использование условных логических выражений в коде;
- изучение ограничений;
- использование конструкций циклов языка Python;

Tlgm: @it_boooks

- функции;
- работа с файлами;
- создание программ на языке Python;
- работа с модулями языка Python.

Теперь можно начинать практическое изучение основ программирования на языке Python.



В этой главе все внимание сосредоточено на изложении основных концепций языка программирования Python для сетевых инженеров, желающих освоить Python для улучшения своих профессиональных навыков. Глава не предназначена для разработчиков ПО, которым необходимо углубленное изучение Python для написания качественных программ для промышленного использования.

Следует также отметить, что концепции, рассматриваемые в данной главе, вполне применимы не только в рамках языка Python. Например, необходимо очень хорошо понять такие общие концепции, как циклы и типы данных, рассматриваемые здесь, чтобы эффективно использовать инструментальные средства, такие как Ansible, Salt, Puppet и StackStorm.

Использование интерактивного интерпретатора Python

Интерактивный интерпретатор Python не очень хорошо известен тем, кто только начинает изучение программирования, и даже тем, кто имеет некоторый опыт разработки на других языках программирования, но мы считаем, что это инструмент, о котором должен знать каждый. Следует изучить этот инструмент, прежде чем приступить к попыткам создания независимо выполняемых скриптов.

Интерпретатор Python представляет собой полезный инструмент для разработчиков любого профессионального уровня. Интерактивный интерпретатор Python, часто называемый командной оболочкой Python shell, используется как учебная платформа для начинающих, но также часто применяется более опытными разработчиками для тестирования и установления обратной связи (получения результатов выполнения) в реальном времени без необходимости написания полной версии программы или скрипта.

Интепретатор Python shell включен практически во все дистрибутивы Linux общего назначения, а также во многие более современные сетевые операционные системы от производителей сетевого оборудования и ПО, таких как Cisco, HP, Juniper, Cumulus, Arista и прочих.

Чтобы начать работу с интерактивным интерпретатором Python, нужно просто открыть окно терминала в Linux или окно сеанса связи по протоколу SSH с современным сетевым устройством, ввести команду `python` и нажать клавишу Enter.



Все примеры в этой главе показаны в виде команд в терминале (командной строке) Linux и начинаются с промпта $. Когда вы находитесь в среде Python shell, все строки и команды начинаются с промпта >>>. Отметим, что все примеры выполнялись в системе Ubuntu 14.04 LTS в версии интерпретатора Python 2.7.6.

Tlgm: @it_boooks

После ввода команды python и нажатия клавиши **Enter** вы сразу попадаете в командную оболочку (shell). В этой командной оболочке можно сразу же писать код на языке Python. Для этого не нужен ни текстовый редактор, ни интегрированная среда разработки (IDE), нет каких-либо особых предварительных требований – можно сразу начать работу.

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Мы будем постепенно рассматривать все подробности программирования на языке Python на протяжении этой главы, но прямо сейчас имеет смысл привести несколько примеров, демонстрирующих мощь интерпретатора Python.

В следующем примере создается переменная hostname и ей присваивается значение ROUTER_1.

```
>>> hostname = 'ROUTER_1'
>>>
```

Отметим отстутствие необходимости предварительного объявления переменной hostname или определения для нее конкретного типа string. Это одно из основных отличий от некоторых широко распространенных языков программирования, таких как C и Java, и именно поэтому Python называется динамическим языком (языком с динамической типизацией).

Выполним вывод значения переменной hostname.

```
>>> print(hostname)
ROUTER_1
>>>
>>> hostname
'ROUTER_1'
>>>
```

После создания переменной ее значение можно без затруднений вывести командой print, но в командной оболочке Python shell также имеется другая возможность вывода значения переменной hostname или любой другой переменной: надо просто ввести имя переменной и нажать клавишу **Enter**. Единственным различием между двумя этими способами является то, что при использовании команды print соответствующим образом интерпретируются управляющие символы, такие как символ перехода на новую строку (\n), а при выводе без команды print интерпретация не выполняется.

В приведенном ниже примере символ \n интерпретируется при использовании команды print и происходит действительный переход на новую строку, но когда после этого вводится только имя переменной и нажимается клавиша **Enter**, то символ \n не интерпретируется, а выводится в составе строки, представляющей значение переменной.

Tlgm: @it_boooks


```
>>> banner = "\n\n   WELCOME TO ROUTER_1   \n\n"
>>>
>>> print(banner)


   WELCOME TO ROUTER_1


>>>
>>> banner
'\n\n   WELCOME TO ROUTER_1   \n\n'
>>>
```

Различие очевидно.

Командная оболочка Python shell представляет собой превосходный инструмент для валидации (проверки корректности) и тестирования. Вероятно, вы заметили, что в предыдущих примерах использовались и одиночные, и двойные кавычки. А можно ли использовать различные типы кавычек в одной строке? Это легко проверить с помощью Python shell.

```
>>> hostname = 'ROUTER_1"
  File "<stdin>", line 1
    hostname = 'ROUTER_1"
                       ^
SyntaxError: EOL while scanning string literal
>>>
```

После проведенной проверки становится понятно, что Python поддерживает оба типа кавычек – одиночные и двойные, но в одной строковой переменной их использовать нельзя.

Большинство примеров в этой главе использует интерпретатор Python, поэтому их можно выполнять и тестировать их поведение.

Продолжим изучение интерпретатора Python и рассмотрим различные типы данных, при этом особое внимание обратим на их применение в сетевой среде.

Типы данных языка Python

В этом разделе представлен обзор различных типов данных языка Python, включая строки, числа (целые и с плавающей точкой), логические значения, списки и словари. Кроме того, кратко описываются кортежи (tuples) и множества (sets).

Разделы, посвященные строкам, спискам и словарям, разделены на две части. В первой части представлено описание самого типа данных, во второй части рассматриваются некоторые методы, встроенные в изучаемый тип данных (built-in methods). Как вы увидите в дальнейшем, методы (methods) являются неотъемлемой частью языка Python и существенно упрощают работу с каждым типом данных.

Например, метод `upper`, который выполняет преобразование букв в строке в верхний регистр, может использоваться в выражении `"router1".upper()` и воз-

Tlgm: @it_boooks

вращает строку ROUTER1. В этой главе вы встретите много примеров, в которых применяются методы.

В разделах, посвященных целым числам и логическим значениям, рассматривается практическое применение математических операторов и логических выражений при написании кода на языке Python.

В конце раздела дается краткое описание кортежей и множеств. Это более сложные типы данных, но даже в пределах вводного курса в Python им необходимо уделить некоторое внимание.

В табл. 4.1 представлены краткие характеристики всех типов данных, рассматриваемых в текущей главе. Эту таблицу можно использовать как мини-справочник по типам данных.

Таблица 4.1. Обзор типов данных Python

Тип данных	Описание	Краткое обозначение типа	Используемые символы	Пример
String (строка)	Последовательность любых символов, заключенная в кавычки	`str`	`""`	`hostname="nycr01"`
Integer (целое число)	Целое число как последовательность десятичных цифр без кавычек	`int`	Нет	`eos_qty=5`
Float (число с плавающей точкой)	Число с плавающей точкой (десятичное)	`float`	Нет	`cpu_util=52.33`
Boolean (логическое значение)	Одно из значений: True или False (без кавычек)	`bool`	Нет	`is_switchport=True`
List (список)	Упорядоченная последовательность значений. Тип значений может быть любым	`list`	`[]`	`vendors=['cisco', 'juniper', 'arista', 'cisco']`
Dictionary (словарь)	Неупорядоченный список пар ключ-значение	`dict`	`{}`	`facts={"vendor":"cisco", "platform":"catalyst", "os":"ios"}`
Set (множество)	Неупорядоченный набор неповторяющихся элементов	`set`	`set()`	`set(vendors)=>['cisco', 'juniper', 'arista']`
Tuple (кортеж)	Упорядоченная и неизменяемая последовательность значений	`tuple`	`()`	`ipaddr=(10.1.1.1, 24)`

Изучение типов данных начнем со строк языка Python.

Использование строк

Строка (string) – это последовательность символов, заключенная в кавычки. Можно предположить, что строка является типом данных, наиболее часто используемым во всех языках программирования.

Tlgm: @it_boooks

Ранее в этой главе рассматривалось несколько простых примеров создания переменных типа string. Но чтобы начать полноценное использование строк, необходимо знать некоторые их свойства.

Сначала определим две новые переменные строкового типа: `final` и `ipaddr`.

```
>>> final = 'The IP address of router1 is: '
>>>
>>> ipaddr = '1.1.1.1'
>>>
```



Можно воспользоваться встроенной функцией языка `type()` для проверки типа данных любого объекта языка Python.

```
>>> type(final)
<type 'str'>
>>>
```

Таким способом можно с легкостью проверить тип любого объекта. Это очень полезно в проблемном коде, особенно если этот код писали не вы.

Далее рассмотрим, как выполняется объединение, наращивание или конкатенация (concatenation) строк.

```
>>> final + ipaddr
'The IP address of router1 is: 1.1.1.1'
```

В этом примере используются две ранее созданные строковые переменные: `final` и `ipaddr`. Они объединяются (concatenate) с помощью оператора +, и результат выводится на экран. Простые и ясные действия.

Операцию объединения строк можно выполнить даже в том случае, если содержимое строковой переменной `final` не является предварительно определенным объектом (то есть представляет собой строковый литерал):

```
>>> print('The IP address of router1 is: ' + ipaddr)
The IP address of router1 is: 1.1.1.1
>>>
```

Использование встроенных методов строк

Для просмотра всех доступных встроенных методов для строк можно воспользоваться встроенной функцией языка dir() непосредственно в командной оболочке Python shell. Сначала нужно создать какую-либо строковую переменную или использовать формальное имя типа str и передать как аргумент в функцию dir(), чтобы получить список всех методов.



Функцию `dir()` можно использовать для любого объекта языка Python, а не только для строк. Мы продемонстрируем использование этой функции для различных типов данных далее в текущей главе.

```
>>>
>>> dir(str)
# output has been omitted
```

Tlgm: @it_boooks


```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'lower',
'lstrip', 'replace', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'upper']
>>>
```

Для подтверждения сказанного выше можно также передать любую строку в функцию `dir()`. Например, если определена переменная `hostname = 'ROUTER'`, то ее имя можно передать в функцию `dir(hostname)`, и будет выведен точно такой же результат, как и при вызове функции `dir(str)`. В обоих случаях мы получаем полный список всех доступных методов для строк.



Функция `dir()` может оказаться чрезвычайно важным средством для определения всех доступных методов конкретного типа данных, поэтому не забывайте про нее.

В показанном выше примере имена некоторых функций содержат начальные и конечные символы подчеркивания. Такие функции не рассматриваются в данной книге, так как наша задача – предоставить практическое введение в язык, а не описывать специальные методы для внутреннего использования (для этого их имена и помечены символами подчеркивания).

Рассмотрим несколько встроенных методов для строк: `count`, `endswith`, `startswith`, `format`, `isdigit`, `join`, `lower`, `upper` и `strip`.



Чтобы узнать, как практически применить конкретный метод из списка, выведенного функцией `dir()`, можно воспользоваться встроенной в язык функцией `help()`. В эту функцию в качестве аргумента передается имя объекта (или переменной) и имя метода. В следующих примерах показаны два способа вызова функции `help()` и выводимая в результате информация об использовании метода `upper`:

```
>>> help(str.upper)
>>>
>>> help(hostname.upper)
>>>
```

В обоих случаях будет выведена следующая информация:

```
Help on method_descriptor:

upper(...)
S.upper() -> string

Return a copy of the string S converted to uppercase.
 (Возвращает копию строки S с преобразованием букв в верхний регистр.)
(END)
```

Чтобы выйти из встроенной системы помощи, нажмите клавишу **Q**.

При изучении каждого метода вы должны найти ответы на два главных вопроса. Какое значение возвращает метод? Какое действие метод выполняет с исходным объектом?

Tlgm: @it_boooks

Использование методов upper() и lower() Методы `upper()` и `lower()` удобны, когда необходимо сравнить строки без учета регистра содержащихся в них букв. Например, требуется принять переменную, содержащую имя интерфейса, такое как «Ethernet1/1», но пользователю также разрешено вводить строку «ethernet1/1». Для решения проблемы несовпадения букв разных регистров можно воспользоваться методом `upper()` или `lower()`.

```
>>> interface = 'Ethernet1/1'
>>>
>>> interface.lower()
'ethernet1/1'
>>>
>>> interface.upper()
'ETHERNET1/1'
>>>
```

Из этого примера понятно, что для вызова метода используется следующий формат: сначала имя объекта или строка, затем точка и имя метода с завершающими круглыми скобками.

Отметим, что после выполнения метода `interface.lower()` в окне терминала выводится строка `ethernet1/1`. Это означает, что при выполнении метода `lower()` возвращается (returned) строка `ethernet1/1`. То же самое относится и к методу `upper()`. Возвращаемое методом значение также можно присвоить новой или существующей переменной.

```
>>> intf_lower = interface.lower()
>>>
>>> print(intf_lower)
ethernet1/1
>>>
```

В этом примере также используется метод `lower()`, но возвращаемое им значение присваивается переменной.

А что произошло с содержимым исходной переменной `interface`? Проверим, не изменилось ли оно.

```
>>> print(interface)
Ethernet1/1
>>>
```

После выполнения первого примера было не вполне понятно, что произошло с содержимым исходной переменной `interface`. После проверки мы убедились в том, что ее содержимое не изменилось – это по-прежнему строка `Ethernet1/1`. Но в дальнейшем мы будем рассматривать и такие примеры, в которых изменяется содержимое исходного объекта после выполнения встроенного метода.

Использование методов startwith() и endswith() По именам этих методов легко понять, что `startswith()` используется для того, чтобы определить, не начинается ли строка с определенной последовательности символов, а с помощью

Tlgm: @it_boooks

endswith() можно проверить, не заканчивается ли строка определенной последовательностью символов.

```
>>> ipaddr = '10.100.20.5'
>>>
>>> ipaddr.startswith('10')
True
>>>
>>> ipaddr.startswith('100')
False
>>>
>>> ipaddr.endswith('.5')
True
>>>
```

В примерах предыдущего раздела методы lower() и upper() возвращали строку с преобразованием в ней букв в нижний и верхний регистр соответственно.

В этом примере метод startswith() возвращает не строку, а объект логического (bool) типа. К логическому типу данных относятся два значения True (истина) и False (ложь). Метод startswith() возвращает значение True, если заданная как аргумент последовательность символов совпадает с начальной последовательностью символов исходного объекта (строки). Метод endswith() проверяет совпадение аргумента с конечной последовательностью символов исходного объекта (строки). Если последовательности не совпадают, возвращается значение False.



Особо отметим, что логическим значением является только True или False без кавычек. Первая буква обязательно должна быть прописной (в верхнем регистре). Более подробно мы рассмотрим логические значения в следующих разделах этой главы.

Методы startswith() и endswith() полезны при проверке начала и конца строк. Это может быть проверка первого или четвертого октета в IPv4-адресе или проверка имени интерфейса, которая в примере предыдущего раздела выполнялась с помощью метода lower(). Вместо того чтобы предполагать, что пользователь скрипта непременно вводит полное имя интерфейса, более удобно проверять только первые два символа, что позволяет пользователю вводить имена «ethernet1/1», «eth1/1» и даже «et1/1».

Для выполнения такой проверки покажем, как можно объединять методы, то есть использовать значение, возвращаемое одним методом, как исходный объект строкового типа для второго метода.

```
>>> interface = 'Eth1/1'
>>>
>>> interface.lower().startswith('et')
True
>>>
```

Этот код проверяет имя Ethernet-интерфейса, выполняя сначала метод lower(), который возвращает строку eth1/1, затем метод startswith() проверяет,

Tlgm: @it_boooks

начинается ли переданная ему строка с последовательности «et». Так как совпадение обнаружено, возвращается логическое значение `True`.

Разумеется, при передаче имени интерфейса как строкового объекта возможны и другие ошибки, не связанные с последовательностью «eth», но здесь важно было продемонстрировать возможность совместной работы двух методов.

Использование метода strip() Многие сетевые устройства пока еще не имеют прикладных программных интерфейсов или API. Поэтому весьма вероятно, что при необходимости написания скрипта для более старых сетевых устройств придется возвращаться к интерфейсу командной строки (CLI) и иметь дело с фрагментами необработанного текста, возвращаемого устройством. Необработанный текст возвращает любая команда `show` – от краткого варианта `show interfaces` до полного `show running-config`.

При необходимости сохранения или вывода на экран содержимого какого-либо объекта нежелательными становятся излишние начальные и конечные пробелы в строках. Для связи с предыдущими примерами предположим, что такой строкой может быть IP-адрес.

Например, необходимо работать со строкой адреса `"  10.1.50.1  "`, содержащей лишние пробелы. Методы `startswith()` и `endswith()` здесь неприменимы, так как не обрабатывают пробелы. Для удаления ненужных пробелов в подобных ситуациях применяется метод `strip()`.

```
>>> ipaddr = '   10.1.50.1   '
>>>
>>>
>>> ipaddr.strip()
'10.1.50.1'
>>>
```

Метод `strip()` возвращает объект (строку) без начальных и конечных пробелов. Здесь не показаны примеры применения `lstrip()` и `rstrip()`, которые также являются встроенными методами для строк и удаляют начальные пробелы слева и конечные пробелы справа соответственно.

Использование метода isdigit() Иногда при работе со строками нужно проверить, не является ли строковый объект числом. С технической точки зрения числа являются совершенно другим типом данных (рассматриваемых в следующих разделах), но цифры как символы могут содержаться и в строковых значениях.

Встроенный метод `isdigit()` позволяет проверить, является ли символ или строка числом, то есть содержит только цифровые символы.

```
>>> ten = '10'
>>>
>>> ten.isdigit()
True
>>>
```

Tlgm: @it_boooks


```
>>> bogus = '10a'
>>>
>>> bogus.isdigit()
False
```

Подобно методам `startswith()` и `endswith()` метод `isdigit()` возвращает логическое значение. Если исходная строка содержит только цифровые символы, то возвращается значение `True`, в противном случае возвращается значение `False`.

Использование метода count() Предположим, что необходимо работать с бинарным числом, например для определения IP-адреса или маски подсети. Конечно, существуют встроенные библиотеки для пребразования двоичного числа в десятичное, но что, если нужно просто подсчитать, сколько 1 или 0 содержится в заданной строке? Для выполнения этой задачи можно воспользоваться встроенным методом `count()`.

```
>>> octet = '11111000'
>>>
>>> octet.count('1')
5
```

В этом примере показано, насколько прост в использовании метод `count()`. В отличие от ранее рассмотренных методов, он возвращает целое числовое значение типа `int` (integer).

Параметр метода `count()` не ограничен одним символом, можно передать в метод строку.

```
>>> octet.count('111')
1
>>>
>>> test_string = "Don't you wish you started programming a little earlier?"
>>>
>>> test_string.count('you')
2
```

Использование метода format() Выше было показано, как объединять строки. Допустим, что нужно создать строку, а еще лучше – команду для передачи в сетевое устройство. Команда также формируется из нескольких строк и переменных. Как правильно отформатировать строку или команду для CLI?

Для следующего примера используем команду `ping` и предположим, что формат этой команды должен быть следующим:

```
ping 8.8.8.8 vrf management
```



В примерах этой главы используются обобщенные (абстрактные) сетевые команды без создания соединений с реальными устройствами. Формат команд не соответствует какому-либо формату команд конкретного производителя оборудования, таким образом, примеры представляют «отраслевой стандарт» и работают на устройствах и в средах различных производителей, включая Cisco IOS, Cisco NXOS, Arista EOS и многие другие.

Tlgm: @it_boooks

Если бы нужно было написать скрипт, то, вероятнее всего, целевой IP-адрес, на который необходимо отправлять ICMP-эхо-запросы, а также узел виртуальной маршрутизации и переадресации (VRF) определялись бы параметрами, вводимыми пользователем. В рассматриваемом здесь примере просто вводятся парметры `'8.8.8.8'` и `'management'` соответственно.

Один из способов формирования требуемой строки начинается с выполнения следующих команд:

```
>>> ipaddr = '8.8.8.8'
>>> vrf = 'management'
>>>
>>> ping = 'ping' + ipaddr + 'vrf' + vrf
>>>
>>> print(ping)
ping8.8.8.8vrfmanagement
```

Но в полученной строке отсутствуют необходимые пробелы, и для исправления этой ситуации существуют два пути – добавить пробелы к вводимым объектам или вставить пробелы непосредственно в формируемый объект `ping`. Рассмотрим первый вариант:

```
>>> ping = 'ping' + ' ' + ipaddr + ' ' + 'vrf ' + vrf
>>>
>>> print(ping)
ping 8.8.8.8 vrf management
```

Этот способ не слишком сложен и вполне успешно работает, но если строки и команды становятся более длинными, то можно быстро запутаться в многочисленных кавычках и пробелах. Использование метода `format()` может упростить задачу.

```
>>> ping = 'ping {} vrf {}'.format(ipaddr, vrf)
>>>
>>> print(ping)
ping 8.8.8.8 vrf management
```

Метод `format()` принимает несколько параметров, которые вставляются вместо пар фигурных скобок ({}), обнаруженных в исходной строке. Отметим, что метод `format()` применяется к необработанным исходным строкам, в отличие от предыдущих примеров.

Встроенные методы строкового типа данных можно использовать и для переменных, и для исходных строк (строковых литералов). Это примечание относится и к встроенным методам других типов данных.

В следующем примере показано использование метода `format()` для предварительно созданного строкового объекта (переменной) в противоположность предыдущему примеру, где обрабатывался строковый литерал.

```
>>> ping = 'ping {} vrf {}'
>>>
```

Tlgm: @it_boooks


```
>>> command = ping.format(ipaddr, vrf)
>>>
>>> print(command)
ping 8.8.8.8 vrf management
```

Такой вариант наиболее вероятен, поскольку применяется предварительно сформированная команда в скрипте на языке Python, пользователь вводит два параметра, а полученная в результате команда передается в сетевое устройство.

Использование методов join() и split() Это последние встроенные методы для строкового типа данных, рассматриваемые в текущей главе. Они рассматриваются в конце раздела, потому что работают не только со строками, но и с другим типом данных – со списком (`list`).



Напомним, что списки подробно рассматриваются в одном из следующих разделов, но здесь необходимо привести очень краткую ознакомительную информацию о списках, чтобы продемонстрировать практическое применение методов `join()` и `split()` для строковых объектов.

Списки в точности соответствуют своему наименованию. Это действительно списки (lists) объектов, при этом каждый объект списка называется элементом (element). Элементы могут быть одного типа или различных типов. Отметим, что совершенно необязательно, чтобы все элементы списка принадлежали к одному типу данных.

Если имеется сетевая среда с пятью маршрутизаторами, то можно сформировать список имен хостов.

```
>>> hostnames = ['r1', 'r2', 'r3', 'r4', 'r5']
```

Также можно создать список команд для передачи в сетевое устройство с целью изменения конфигурации. В следующем примере приведен список команд для корректного завершения работы Ethernet-интерфейса на коммутаторе.

```
>>> commands = ['config t', 'interface Ethernet1/1', 'shutdown']
```

Создание списков, подобных этому, является общепринятой практикой, но если используется обычное сетевое устройство с интерфейсом командной строки, то не всегда имеется возможность передать объект типа `list` непосредственно в нужное устройство. Для передачи в устройство может потребоваться формат символьной строки (или отдельно передаваемые команды).

Метод `join()` в качестве параметра принимает список и создает (возвращает) строку, при этом по необходимости вставляет требуемые символы между элементами списка.

Напомним, что символ \n обозначает конец текущей строки и переход на новую строку. При передаче команд в устройство может потребоваться вставка символа \n между командами, чтобы позволить устройству выполнять каждую очередную команду в отдельной строке.

Tlgm: @it_boooks

Используем список команд commands из предыдущего примера и рассмотрим практическое применение метода join() для создания одной строки с вставленными в нее между командами символами \n.

```
>>> '\n'.join(commands)
'config t\ninterface Ethernet1/1\nshutdown'
>>>
```

Другой практический пример – использование API, например NX-API для коммутаторов Cisco Nexus. Cisco предоставляет возможность передавать строку команд, но эти команды должны разделяться символом точки с запятой (;).

Для выполнения этой задачи применим тот же подход.

```
>>> ' ; '.join(commands)
'config t ; interface Ethernet1/1 ; shutdown'
>>>
```

В этом примере добавлены пробелы до и после точки с запятой, но в целом способ применения тот же самый.



В показанных выше примерах точка с запятой и символ перехода на новую строку использовались как разделители, но следует также знать, что разделители не обязательны. Можно объединять элементы из списка без вставки между ними каких-либо символов, например: ''.join(list).

Теперь вы знаете, как использовать метод join() для формирования строк из списков, но что, если нужно выполнить противоположную задачу, то есть создать список из строки? Одним из решений является применение метода split().

В следующем примере воспользуемся строкой, сгенерированной в предыдущем примере, и преобразуем ее обратно в список.

```
>>> commands = 'config t ; interface Ethernet1/1 ; shutdown'
>>>
>>> cmds_list = commands.split(' ; ')
>>>
>>> print(cmds_list)
['config t', 'interface Ethernet1/1', 'shutdown']
>>>
```

Этот пример показывает, как просто можно превратить строку в список. Другой пример в сетевом контексте – преобразование исходной строки IP-адреса в список из четырех элементов – по количеству октетов в строке.

```
>>> ipaddr = '10.1.20.30'
>>>
>>> ipaddr.split('.')
['10', '1', '20', '30']
>>>
```

В этом разделе были описаны основы работы со строками языка Python. В следующем разделе рассматривается следующий тип данных – числовые значения.

Tlgm: @it_boooks

Использование числовых значений

Мы не будем тратить время на изучение разнообразных типов числовых значений, таких как числа с плавающей (десятичной) точкой или мнимые числа, а кратко рассмотрим тип данных, который обозначен как `int`, то есть целое число (integer). Откровенно говоря, причина такого ограничения в том, что для нецелых чисел нет понятных встроенных методов, которые заслуживали бы внимания в контексте этой книги (то есть для сетевой среды). В то же время для целых чисел существуют простые и понятные встроенные методы и математические операторы, выполняемые непосредственно в командной оболочке Python shell. Эти методы и операторы мы и будем рассматривать.



Следует помнить, что десятичные (нецелые) числа во многих языках программирования, в том числе и в Python, обозначаются как числа с плавающей точкой (floats). Напомним также, что тип данных можно проверить с помощью встроенной функции языка `type()`:

```
>>> cpu = 41.3
>>>
>>> type(cpu)
<type 'float'>
>>>
>>>
```

Выполнение математических операций

Для сложения чисел не требуется никаких экстраординарных действий – все делается очень просто.

```
>>> 5 + 3
8
>>> a = 1
>>> b = 2
>>> a + b
3
```

Иногда необходим счетчик в цикле обработки последовательности объектов. В начале цикла определяется счетчик `counter = 1`, выполняются некоторые операции, затем счетчик увеличивается на единицу `counter = counter + 1`. Это работает вполне эффективно, но в Python есть специальный оператор для таких случаев: `counter += 1` (его называют оператором инкремента). Применение этого оператора показано в следующем примере.

```
>>> counter = 1
>>> counter = counter + 1
>>> counter
2
>>>
>>> counter = 5
>>> counter += 5
>>>
>>> counter
10
```

Tlgm: @it_boooks

Вычитание выполняется в том же стиле, что и сложение. Поэтому сразу переходим к практическому примеру.

```
>>> 100 - 90
10
>>> count = 50
>>> count - 20
30
>>>
```

При умножении также не встречается никаких неожиданностей – вот простой пример.

```
>>> 100 * 50
5000
>>>
>>> print(2 * 25)
50
>>>
```

Полезной функциональной особенностью оператора умножения (*) является возможность его использования для строк. Оператор умножения позволяет выполнять некоторое форматирование и добавлять своеобразные «украшения» в выводимые строки.

```
>>> print('*' * 50)
**************************************************
>>>
>>> print('=' * 50)
==================================================
>>>
```

Приведенный пример предельно прост, но в то же время демонстрирует интересные возможности применения оператора умножения в строках. Не зная об этой возможности, пользователи обычно склонны применять команды вывода простейшим образом с набором длиной последовательности одинаковых символов в одной строке: `print(**********)`. После этой подсказки и нескольких других, рассматриваемых ниже в этой главе, аккуратно оформленный вывод текстовых данных выполняется гораздо проще.

Если вы не занимались вычислениями вручную в последние несколько лет, то операция деления может выглядеть несколько затруднительной. Но в языке Python деление практически ничем не отличается от трех других арифметических операций и выполняется аналогично.

Точно так же вы просто вводите то, что хотите вычислить. Вот простые примеры выполнения операции деления:

```
>>> 100 / 50
2
>>>
>>> 10/ 2
5
>>>
```

Tlgm: @it_boooks

Результаты вполне ожидаемы и очевидны.

Отличие от прочих арифметических операций проявляется лишь при делении с остатком:

```
>>> 12 / 10
1
>>>
```

Понятно, что число 10 «укладывается» в числе 12 только один раз. Количество таких «укладок» называют частным (quotient), и в нашем примере частное равно 1. Та часть, которая не возвращается при делении и не выводится на экран, называется остатком (remainder). Чтобы получить остаток средствами языка Python, необходимо воспользоваться оператором `%`, то есть оператором вычисления остатка от деления (или деления по модулю).

```
>>> 12 % 10
2
>>>
```

Для вычисления полного конечного результата деления используются оба оператора / и `%`.

На этом краткий обзор средств вычислений в языке Python окончен. Переходим к работе с логическими значениями.

Использование логических значений

Логические объекты или объекты типа `bool` в языке Python весьма просты. Сначала рассмотрим основы общей булевой логики, объединенные в таблицу вычисления истинных и ложных значений (табл. 4.2).

Таблица 4.2. Таблица вычисления логических значений истина/ложь

A	B	A and B	A or B	Not A
False	**False**	False	False	True
False	**True**	False	True	True
True	**False**	False	True	False
True	**True**	True	True	False

Отметим, что все значения в таблице представляют собой либо `True` (истина), либо `False` (ложь). В булевой логике все возможные значения сведены к паре противоположных значений `True` или `False`. Поэтому основы логики легко понять.

Поскольку логические значения представлены только как `True` или `False`, результатами вычисления логических выражений также могут быть только эти два значения. Из приведенной выше таблицы следует, что для получения значения `True` в выражении `A and B` оба операнда A и B обязательно должны содержать значения `True`. Выражение `A or B` дает результат `True`, если один из операндов (A или B) представлен значением `True`. Также очевидно, что применение

Tlgm: @it_boooks

операции логического отрицания NOT инвертирует значение, то есть изменяет его на противоположное. Не вызывает никаких сомнений тот факт, что выражение NOT False дает результат True, а результатом выражения NOT True является False.

В языке Python применяются те же самые логические правила. Существуют только два логических значения True и False. Для присваивания одного из этих значений нужно ввести команду, показанную ниже (в логических значениях первая буква обязательно должна быть прописной, а сами значения указываются без кавычек).

```
>>> exists = True
>>>
>>> exists
True
>>>
>>> exists = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>>
```

Из этого примера понятно, что присваивание логического значения выполняется предельно просто. Благодаря наличию обратной связи в реальном времени в интерпретаторе Python мы сразу же видим, что использование буквы нижнего регистра t во время присваивания логического значения True приводит к ошибке, так как значение true неизвестно интерпретатору.

Ниже приводится еще несколько примеров применения логических выражений в интерпретаторе Python.

```
>>> True and True
True
>>>
>>> True or False
True
>>>
>>> False or False
False
>>>
```

В следующем примере вычисляются те же логические выражения, но с переменными, которым присваиваются логические значения.

```
>>> value1 = True
>>> value2 = False
>>>
>>> value1 and value2
False
>>>
>>> value1 or value2
True
>>>
```

Tlgm: @it_boooks

Отметим, что в логических выражениях может быть больше двух объектов (операндов).

```
>>> value3 = True
>>> value4 = True
>>>
>>> value1 and value2 and value3 and value4
False
>>>
>>> value1 and value3 and value4
True
>>>
```

При получении информации от сетевых устройств логические значения часто используются для быстрой проверки. Является ли интерфейс маршрутизируемым портом? Возможно ли конфигурирование управления интерфейсом? Доступно ли устройство? Для ответа на эти вопросы операция может быть весьма сложной, но в любом случае результатом будет либо `True`, либо `False`.

Вопрос может иметь противоположное значение: является ли недоступным интерфейс, коммутируемый порт или устройство? Нет никакого смысла создавать переменные или объекты для каждого вопроса, но можно воспользоваться логическим оператором `not`, точно зная, что операция с применением `not` возвращает инвертированное логическое значение.

Рассмотрим пример практического применения оператора `not`.

```
>>> not False
>>> True
>>>
>>> is_layer3 = True
>>> not is_layer3
False
>>>
```

В этом примере используется переменная `is_layer3`. Ей присваивается значение `True`, обозначающее, что некоторый интерфейс является портом уровня 3. Если мы применяем выражение `not is_layer3`, то полученный результат позволяет узнать, что это порт уровня 2.

Немного позже в этой главе мы рассмотрим условные выражения (с использованием конструкций `if-else`), но на основе простой логики вы можете узнать о том, действительно ли интерфейс принадлежит уровню 3. Для этого применяется выражение типа `if is_layer3:`, но если необходимо выполнить какое-либо действие, если это интерфейс уровня 2, то выражение должно быть таким: `if not is_layer3:`.

В дополнение к операторам `and` и `or` для генерации логических объектов (значений) используются операторы `==` (равно) и `!=` (не равно). С помощью этих операторов можно выполнять сравнения или проверки, чтобы узнать, равны или не равны объекты (в одном выражении можно объединять с помощью скобок проверки на равенство больше двух объектов).

Tlgm: @it_boooks


```
>>> True == True
True
>>>
>>> True != False
True
>>>
>>> 'network' == 'network'
True
>>>
>>> 'network' == 'no_network'
False
>>>
```

После беглого обзора способов работы с логическими объектами, операндами и выражениями можно перейти к рассмотрению использования списков в языке Python.

Использование списков

Очень краткое введение в использование списков было представлено при изучении встроенных в строки методов `join()` и `split()`. Здесь мы рассмотрим списки более подробно.

Список – это объект типа `list`, простейшим уровнем представления которого является упорядоченная последовательность объектов. Напомним простые примеры из предыдущего раздела главы, в котором рассматривался метод `join()` для строк, чтобы еще раз показать, как создается список. В этих примерах создаются списки строк, но возможны списки, состоящие из элементов любых других типов данных, как будет продемонстрировано ниже.

```
>>> hostnames = ['r1', 'r2', 'r3', 'r4', 'r5']
>>> commands = ['config t', 'interface Ethernet1/1', 'shutdown']
>>>
```

В следующем примере показан список объектов, каждый из которых принадлежит к различным типам данных.

```
>>> new_list = ['router1', False, 5]
>>>
>>> print(new_list)
['router1', False, 5]
>>>
```

Из приведенных примеров становится понятно, что списки представляют собой упорядоченную последовательность объектов, заключенную в квадратные скобки. При работе со списками наиболее часто выполняется операция доступа к отдельному элементу списка.

Создадим новый список интерфейсов и покажем, как можно вывести на экран отдельный элемент списка.

```
>>> interfaces = ['Eth1/1', 'Eth1/2', 'Eth1/3', 'Eth1/4']
>>>
```

Tlgm: @it_boooks

После создания списка три его элемента выводятся поочередно.

```
>>> print(interfaces[0])
Eth1/1
>>>
>>> print(interfaces[1])
Eth1/2
>>>
>>> print(interfaces[2])
Eth1/3
>>>
```

Для доступа к отдельным элементам списка используется значение индекса (index) требуемого элемента, заключенное в квадратные скобки. Здесь важно отметить, что индекс начинается с 0, а последнее значение индекса вычисляется по схеме «длина списка минус 1». В нашем примере для доступа к первому элементу используется выражение `interfaces[0]`, а для доступа к последнему элементу – `interfaces[3]`.

В том же примере сразу понятно, что список состоит из четырех элементов, но что, если длина списка неизвестна?

На этот случай в Python предусмотрена встроенная функция `len()`.

```
>>> len(interfaces)
4
>>>
```

Для доступа к самому последнему элементу списка можно применить еще один способ: `list[-1]`.

```
>>> interfaces[-1]
'Eth1/4'
>>>
```



Часто термины функция (function) и метод (method) используются как взаимозаменяемые, но до сего момента мы рассматривали главным образом методы, а не функции. Небольшое различие состоит в том, что функция вызывается без ссылки на родительский объект. Как вы видели ранее, для вызова встроенного метода объекта применяется синтаксис `object.method()`. Функция вызывается просто по имени, например `len()`. Тем не менее очень часто метод называют функцией.

Использование встроенных методов списков

Для просмотра всех доступных встроенных методов для списков используется та же функция `dir()`, которая ранее применялась по отношению к объектам типа строка. Можно создать любую переменную типа список или воспользоваться формальным именем типа данных `list` и передать его как аргумент в функцию `dir()`. Мы используем созданный в предыдущем разделе список `interfaces`.

```
>>> dir(interfaces)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Tlgm: @it_boooks



Чтобы пример выглядел более простым и понятным, из выведенного списка были удалены все объекты, в именах которых имеются начальные и конечные символы подчеркивания.

Рассмотрим практическое использование некоторых встроенных методов.

Использование метода append() Важным свойством имен методов является то, что они удобны для чтения человеком и интуитивно понятны в большинстве случаев. Метод `append()` используется для добавления (append) элемента в существующий список.

Мы продемонстрируем эту операцию в следующем примере, но начнем с создания пустого списка. Для этого объекту присваиваются «пустые» квадратные скобки (то есть между скобками нет никаких символов).

```
>>> vendors = []
>>>
```

Теперь добавим наименования производителей в этот список.

```
>>> vendors.append('arista')
>>>
>>> print(vendors)
['arista']
>>>
>>> vendors.append('cisco')
>>>
>>> print(vendors)
['arista', 'cisco']
>>>
```

Из примера понятно, что метод `append()` добавляет элемент в последнюю позицию списка, то есть в конец списка. В отличие от многих методов для строковых объектов, этот метод не возвращает никакого значения, но изменяет исходную переменную или объект.

Использование метода insert() Может потребоваться не только добавление элемента к концу списка, но и вставка (insert) элемента в заданную позицию списка. Для выполнения этой операции предназначен метод `insert()`.

В метод `insert()` необходимо передать два аргумента. Первый аргумент – номер позиции или индекс, по которому должен быть вставлен новый элемент, второй аргумент – сам объект, вставляемый в список.

Для следующего примера сначала создадим список команд.

```
>>> commands = ['interface Eth1/1', 'ip address 1.1.1.1/32']
```



Напомним, что команды в приводимых примерах являются обобщенными и не соответствуют в полной мере командам какого-либо конкретного производителя или платформы.

Теперь предположим, что необходимо добавить еще две команды в список `['interface Eth1/1', 'ip address 1.1.1.1/32']`. В качестве самого первого элемента

Tlgm: @it_boooks

необходимо добавить команду `config t`, а перед IP-адресом нужно добавить `no switchport`.

```
>>> commands = ['interface Eth1/1', 'ip address 1.1.1.1/32']
>>>
>>> commands.insert(0, 'config t')
>>>
>>> print(commands)
['config t', 'interface Eth1/1', 'ip address 1.1.1.1/32']
>>>
>>> commands.insert(2, 'no switchport')
>>>
>>> print(commands)
['config t', 'interface Eth1/1', 'no switchport', 'ip address 1.1.1.1/32']
>>>
```

Использование метода count() При выполнении инвентаризации типов устройств в сети можно сформировать список, в котором некоторые объекты будут содержаться неоднократно. Для следующего примера расширим список наименований производителей сетевых устройств из предыдущего раздела следующим образом:

```
>>> vendors = ['cisco', 'cisco', 'juniper', 'arista', 'cisco', 'hp', 'cumulus', 'arista',
'cisco']
>>>
```

Теперь можно подсчитать (count), сколько экземпляров заданного объекта содержится в этом списке с помощью метода `count()`. В приведенном ниже примере определяется, сколько устройств Cisco и Arista используется в рассматриваемой сетевой среде.

```
>>> vendors.count('cisco')
4
>>>
>>> vendors.count('arista')
2
>>>
```

Отметим, что метод `count()` возвращает целое число (тип `int`) и не изменяет существующий исходный объект, в отличие от методов `insert()`, `append()` и некоторых других методов, рассматриваемых в следующих разделах.

Использование методов pop() и index() Практически все рассмотренные до сих пор методы либо изменяли исходный объект, либо возвращали некоторое значение. Метод `pop()` делает и то, и другое.

```
>>> hostnames = ['r1', 'r2', 'r3', 'r4', 'r5']
>>>
```

Здесь приведен список имен хостов. Допустим, что устройство, которому соответствует имя хоста `r5`, удаляется из сети, поэтому необходимо быстро исключить (вытолкнуть – pop) это имя из списка.

Tlgm: @it_boooks


```
>>> hostnames.pop()
'r5'
>>>
>>> print(hostnames)
['r1', 'r2', 'r3', 'r4']
>>>
```

В примере показано, что метод возвращает элемент, удаляемый (выталкиваемый) из списка, а сам список изменяется.

Кроме того, вы, вероятнее всего, обратили внимание на то, что в метод не передается ни элемент, ни индекс, таким образом, по умолчанию метод `pop()` удаляет самый последний элемент списка.

А если нужно удалить элемент `r2`? Чтобы удалить элемент, который не является последним в списке, необходимо передать в метод значение индекса удаляемого элемента. Но как определить индексное значение элемента? Для этого предназначен метод `index()`.

Ниже приведен пример определения значения индекса для конкретного заданного элемента.

```
>>> hostnames.index('r2')
1
>>>
```

Теперь мы знаем, что элементу `r2` соответствует индекс `1`.

Таким образом, чтобы удалить элемент `r2` из списка, необходимо выполнить следующую операцию:

```
>>> hostnames.pop(1)
'r2'
>>>
>>> print(hostnames)
['r1', 'r3', 'r4']
>>>
```

Определение индекса и удаление соответствующего элемента можно выполнить в одно действие:

```
hostnames.pop(hostnames.index('r2'))
```

Использование метода sort() Последним встроенным методом списков, который мы рассмотрим в текущем разделе, является метод `sort()`. По имени легко понять, что метод выполняет сортировку списка.

В приведенном ниже примере имеется неупорядоченный список доступных IP-адресов, а метод `sort()` используется для обновления исходного объекта. Отметим, что метод не возвращает какое-либо значение.

```
>>> available_ips
['10.1.1.1', '10.1.1.9', '10.1.1.8', '10.1.1.7', '10.1.1.4']
>>>
>>>
>>> available_ips.sort()
```

Tlgm: @it_boooks


```
>>>
>>> available_ips
['10.1.1.1', '10.1.1.4', '10.1.1.7', '10.1.1.8', '10.1.1.9']
```



Следует обратить особое внимание на то, что в предыдущем примере выполняется сортировка IP-адресов как строк, а не как числовых значений.

Почти во всех примерах, в которых используются списки, элементами списков были объекты одного и того же типа, то есть рассматривались списки строк команд, IP-адресов, производителей оборудования, имен хостов. Но если нужно создать список, состоящий из объектов различных типов (и даже из элементов с различными типами данных), не возникает никаких затруднений.

Простой пример хранения объектов различных типов в одном списке возникает при необходимости сохранения информации о конкретном устройстве. Предположим, что нужно хранить имя хоста, наименование производителя устройства и версию ПО или ОС. Список для хранения таких атрибутов устройства может выглядеть следующим образом:

```
>>> device = ['router1', 'juniper', '12.2']
>>>
```

Поскольку элементы в списке индексируются целыми числами, необходимо помнить, какой числовой индекс соответствует каждому атрибуту. В приведенном примере такое соответствие запоминается без труда, но что, если потребуется обеспечить доступ к 10, 20 или 100 атрибутам? Дополнительная трудность возникает из-за того, что списки упорядочены. Поэтому замену, обновление или перемещение любого элемента в списке следует выполнять чрезвычайно осторожно.

Было бы гораздо лучше, если бы существовала возможность обращаться к отдельным элементам списка по имени, а не запоминать точный порядок расположения последовательности элементов. То есть вместо получения имени хоста с помощью выражения `device[0]` использовать что-то, похожее на `device['hostname']`.

В языке Python именно так организована структура словарей – следующего типа данных, который мы рассматриваем в этой главе.

Использование словарей

Мы уже рассмотрели некоторые общеизвестные типы данных, такие как строки, целые числа, логические значения и списки, которые существуют практически во всех языках программирования. В этом разделе речь пойдет о словарях, которые представляют собой особый тип данных в языке Python. В других языках такой тип данных обозначается как ассоциативные массивы, отображения (maps) или хеши (hash maps).

Словари (dictionaries) – это неупорядоченные списки, в которых доступ к значениям (values) осуществляется по именам или по ключам (keys), а не по

Tlgm: @it_boooks

целочисленному индексу. Словари представляют собой набор неупорядоченных пар ключ-значение (key-value), называемых элементами (items).

Предыдущий раздел завершался следующим примером списка:

```
>>> device = ['router1', 'juniper', '12.2']
>>>
```

Если в этом примере заменить список словарем, то получим такой результат:

```
>>> device = {'hostname': 'router1', 'vendor': 'juniper', 'os': '12.1'}
>>>
```

Схема формирования словаря: открывающая фигурная скобка ({), ключ, двоеточие, значение, пары ключ-значение разделяются запятыми (,) в конце – закрывающая фигурная скобка (}).

После создания объекта `dict` доступ к требуемому значению осуществляется с помощью выражения *`dict[key]`*.

```
>>> print(device['hostname'])
router1
>>>
>>> print(device['os'])
12.1
>>>
>>> print(device['vendor'])
juniper
>>>
```

Как уже было сказано выше, словари неупорядочены, в отличие от упорядоченных списков. Это можно наблюдать при выводе содержимого словаря `device` в следующем примере. Пары ключ-значение выводятся не в том порядке, в котором они записывались при создании словаря.

```
>>> print(device)
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
```

Следует отметить, что словарь, показанный в предыдущем примере, можно создать несколькими различными способами. Два способа показаны в следующих блоках кода.

```
>>> device = {}
>>> device['hostname'] = 'router1'
>>> device['vendor'] = 'juniper'
>>> device['os'] = '12.1'
>>>
>>> print(device)
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
```

```
>>> device = dict(hostname='router1', vendor='juniper', os='12.1')
>>>
```

Tlgm: @it_boooks


```
>>> print(device)
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
```

Использование встроенных методов словарей

Для словарей в языке Python существует набор встроенных методов, заслуживающих внимания, и мы, как обычно, рассмотрим самые полезные из них.

Как и для других типов данных, в первую очередь выводится список всех доступных методов, за исключением «методов для внутреннего пользования», имена которых начинаются и заканчиваются символами подчеркивания.

```
>>> dir(dict)
['clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys',
'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values',
'viewitems', 'viewkeys', 'viewvalues']
>>>
```

Использование метода get() Выше был показан способ доступа к паре ключ-значение в словаре с помощью выражения *dict[key]*. Такой способ часто применяется, но у него есть один недостаток. Если ключ не существует, то возникает ошибка KeyError.

```
>>> device
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
>>> print(device['model'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'model'
>>>
```

Метод `get()` предоставляет другой способ доступа, более безопасный для тех случаев, когда прерывание выполнения при отсутствии ключа с выводом сообщения об ошибке нежелательно.

Сначала рассмотрим случай с использованием метода `get()`, когда ключ существует.

```
>>> device.get('hostname')
'router1'
>>>
```

Теперь посмотрим, что происходит, если ключ не существует:

```
>>> device.get('model')
>>>
```

Из предыдущего примера становится понятно, что при отсутствии ключа в словаре метод не возвращает никакого значения. Во многих случаях такой вариант лучше, чем вывод сообщения об ошибке и останов выполнения. Кроме того, метод `get()` позволяет пользователю определить значение, возвращаемое в том случае, когда заданный ключ не существует. Например:

Tlgm: @it_boooks


```
>>> device.get('model', False)
False
>>>
>>> device.get('model', 'DOES NOT EXIST')
'DOES NOT EXIST'
>>>
>>>
>>> device.get('hostname', 'DOES NOT EXIST')
'router1'
>>>
```

Как видите, все очень просто: значение, указанное справа от ключа, возвращается в том и только в том случае, если этого ключа нет в словаре.

Использование методов keys() и values() Словарь – это неупорядоченный список пар ключ-значение. С помощью методов `keys()` и `values()` можно отдельно вывести списки ключей и значений. При вызове каждого метода выводится соответствующий его имени список (`list`) ключей или значений, содержащихся в словаре.

```
>>> device.keys()
['os', 'hostname', 'vendor']
>>>
>>> device.values()
['12.1', 'router1', 'juniper']
>>>
```

Использование метода pop() Встроенный метод `pop()` уже рассматривался ранее, в разделе, посвященном спискам. Точно такой же метод существует и для словарей, и используется он весьма похожим образом. Для списков в метод `pop()` передавался числовой индекс удаляемого элемента, для словарей передается ключ.

```
>>> device
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
>>> device.pop('vendor')
'juniper'
>>>
>>> device
{'os': '12.1', 'hostname': 'router1'}
>>>
```

В приведенном примере можно видеть, что метод `pop()` изменяет исходный объект и возвращает удаляемое («выталкиваемое») значение (value).

Использование метода update() Иногда возникает необходимость добавления некоторой информации к данным, хранящимся в словаре языка Python (в наших примерах это имя хоста, производитель устройства, версия ПО или ОС), то есть обновления (update) словаря с помощью другого словаря, содержащего дополнительные атрибуты устройства.

Tlgm: @it_boooks

Сначала создадим два различных словаря.

```
>>> device
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper'}
>>>
>>> oper = dict(cpu='5%', memory='10%')
>>>
>>> oper
{'cpu': '5%', 'memory': '10%'}
>>>
```

После этого метод `update()` можно применить для обновления одного из словарей, просто добавляя содержимое одного словаря в другой, например содержимое `oper` в словарь `device`.

```
>>> device.update(oper)
>>>
>>> print(device)
{'os': '12.1', 'hostname': 'router1', 'vendor': 'juniper', 'cpu': '5%', 'memory': '10%'}
>>>
```

Метод `update()` не возвращает никаких значений. Изменяется только сам исходный обновляемый объект, в данном примере `device`.

Использование метода items() При работе со словарями метод `items()` используется многократно, поэтому чрезвычайно важно понять, как он работает, разумеется, не в ущерб понимания функциональности других методов.

Ранее мы наблюдали, как осуществляется доступ к отдельным значениям с помощью метода `get()`, как получить список ключей и значений, используя методы `keys()` и `values()` соответственно.

Но как получить доступ к конкретной паре ключ-значение, то есть к определенному элементу (item) словаря, и как выполнить последовательное обращение ко всем элементам? Если необходим итеративный проход (то есть цикл) по словарю и одновременный доступ к ключам и значениям, то наиболее удобным инструментом для выполнения этой задачи становится метод `items()`.



Несколько позже в этой главе будет приведена вводная информация по практическому использованию циклов, но поскольку метод `items()` весьма часто применяется в цикле `for`, здесь приводится пример такого цикла. В приведенном ниже примере важно понять не столько суть работы цикла (она будет рассматриваться в соответствующем разделе ниже), сколько то, что при использовании цикла `for` с применением метода `items()` можно одновременно получить доступ к ключу и значению конкретного элемента словаря.

Самый простой пример – последовательный проход по словарю с помощью оператора цикла `for` и вывод ключа и значения каждого элемента. Еще раз отметим, что циклы подробно рассматриваются в одном из следующих разделов, но здесь цикл необходим для понимания работы метода `items()`.

```
>>> for key, value in device.items():
...     print(key + ': ' + value)
```

Tlgm: @it_boooks


```
...
os :  12.1
hostname :  router1
vendor :  juniper
cpu :  5%
memory :  10%
>>>
```

Здесь важно отметить, что переменные `key` и `value` для цикла `for` определяются пользователем и могут иметь любые имена, как показано в следующем примере.

```
>>> for my_attribute, my_value in device.items():
...     print(my_attribute + ': ' + my_value)
...
os :  12.1
hostname :  router1
vendor :  juniper
cpu :  5%
memory :  10%
>>>
```

Итак, мы рассмотрели основные типы данных языка Python. Теперь вы хорошо понимаете, как нужно работать со строками, числами, логическими значениями, списками и словарями. Далее приводится краткая ознакомительная информация еще о двух типах данных – множествах и кортежах, которые представляют собой чуть более сложные типы данных, по сравнению с изученными ранее.

Множества и кортежи языка Python

Следующие два типа данных можно было бы и не рассматривать при первом знакомстве с языком Python, но в начале главы было отмечено, что мы намеренно включили краткий обзор этих типов, для того чтобы дать полноценное представление о типах данных в языке Python. Этими типами являются множество `set` и кортеж `tuple`.

Если вы поняли работу со списками, то без затруднений поймете работу со множествами. Множества (sets) – это списки элементов, но во множестве может содержаться только один экземпляр каждого элемента, а все дополнительные экземпляры этого элемента не могут быть проиндексированы (то есть к ним невозможно получить доступ по индексу, как в списках).

Внешне множество выглядит как список, но этот список включен в структуру с ключевым словом `set()`:

```
>>> vendors = set(['arista', 'cisco', 'arista', 'cisco', 'juniper', 'cisco'])
>>>
```

В приведенном примере показано множество, в котором при создании содержалось несколько одинаковых элементов. Похожий пример демонстрировал применение метода `count()` для списков, где нужно было подсчитать,

Tlgm: @it_boooks

сколько устройств конкретного производителя используется в сети. Но если необходимо только узнать, устройства каких производителей имеются в сетевой среде и сколько этих (наименований) производителей, то можно воспользоваться множеством.

```
>>> vendors = set(['arista', 'cisco', 'arista', 'cisco', 'juniper', 'cisco'])
>>>
>>> vendors
set(['cisco', 'juniper', 'arista'])
>>>
>>> len(vendors)
3
>>>
```

Отметим, что множество `vendors` содержит только три элемента.

В следующем примере показано, что происходит при попытке доступа к конкретному элементу во множестве. Чтобы получить доступ к элементам множества, необходимо выполнить итеративный проход по всем элементам, например с помощью цикла `for`.

```
>>> vendors[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>>
```

В качестве «домашнего задания» читателю предлагается самостоятельно изучить встроенные методы для множеств.

Кортеж (tuple) – весьма любопытный тип данных. Наилучший способ понять его – сравнение со списком. Как нам уже известно, списки являются изменяемыми (mutable) структурами, то есть можно обновлять, расширять и изменять их любым другим способом. В отличие от списков, кортежи – неизменяемые (immutable) структуры, их нельзя изменить после того, как они созданы. Но, как и в списках, возможен доступ к отдельным элементам кортежа.

```
>>> description = tuple(['ROUTER1', 'PORTLAND'])
>>>
>>>
>>> description
('ROUTER1', 'PORTLAND')
>>>
>>>
>>> print(description[0])
ROUTER1
>>>
```

После создания объекта `description` изменить его невозможно. Нельзя изменить какой-либо элемент, нельзя добавить новые элементы. Это может оказаться полезным, если необходимо, чтобы после создания объекта ни функция, ни пользователь не имели возможности каким-либо образом изменить его.

Tlgm: @it_boooks

В следующем примере наглядно показана невозможность изменения кортежа, кроме того, у кортежа нет таких методов, как update() и append().

```
>>> description[1] = 'trying to modify one'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'count', 'index']
>>>
```

Для сравнения общих свойств и выделения различий между списками, кортежами и множествами ниже приводится краткий итоговый обзор этих типов данных:

- списки изменяемы, их можно корректировать, возможен прямой доступ к отдельным элементам, разрешены повторяющиеся значения;
- множества изменяемы, их можно корректировать, прямой доступ к отдельным элементам невозможен, повторяющиеся значения не допускаются;
- кортежи неизменяемы, после создания их нельзя обновлять и дополнять, возможен прямой доступ к отдельным элементам, разрешены повторяющиеся значения.

На этом раздел типов данных завершается. Теперь вы достаточно хорошо знакомы с рассмотренными здесь типами данных, включая строки, числовые и логические значения, списки, словари, множества и кортежи.

Далее будут рассматриваться условные (логические) выражения (конструкции if then else) в языке Python.

Использование условных логических выражений

Глубокое понимание различных типов данных и объектов и умение работать с ними необходимо. Но богатые возможности программирования ярче всего проявляются, когда вы начинаете использовать эти объекты в логических конструкциях кода, например для выполнения некоторой задачи или для создания объекта, если определенное условие выполняется, то есть дает результат «истина» (или наоборот – не выполняется, то есть дает результат «ложь»).

Условные выражения являются главной составляющей логики, применяемой в коде. Изучение условных выражений начнем с оператора if.

Рассмотрим простой пример, в котором проверяется значение строки.

```
>>> hostname = 'NYC'
>>>
```

Tlgm: @it_boooks


```
>>> if hostname == 'NYC':
...     print('The hostname is NYC')
...
The hostname is NYC
>>>
```

Даже если вы не знали языка Python до начала чтения текущей главы, вероятнее всего, вы поймете смысл приведенного примера. Именно в этом заключается преимущество практического использования Python – обеспечение удобства для чтения человеком, насколько это возможно.

Следует особо отметить две особенности синтаксиса, используемого при работе с оператором `if`. Во-первых, все условные выражения `if` завершаются символом двоеточия (`:`). Во-вторых, код, выполняемый в том случае, когда условие истинно, является частью смещенного вправо блока кода – это обязательное смещение должно иметь размер в четыре пробела (по соглашению), но с технической точки зрения размер смещения не имеет значения. Гораздо важнее сохранять логическую согласованность и всегда использовать одинаковый размер отступа блока в своем коде (опять же с технической точки зрения).



Вообще говоря, при написании кода на языке Python использование смещения размером в четыре пробела – наилучший вариант. Это соглашение принято всем сообществом пользователей языка Python как неофициальный стандарт Python-кода. Такое соглашение существенно упрощает распространение и совместную разработку кода.

В следующем примере показан весь блок кода, смещенный вправо.

```
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
...     print(len(hostname))
...     print('The End.')
...
This hostname is NYC
3
The End.
>>>
```

Теперь, когда понятно, как сформировать условное выражение с оператором if, рассмотрим эту конструкцию более подробно.

Если необходимо проверить, является ли именем хоста не только «NYC», но и «NJ», то следует воспользоваться оператором `else if` или его сокращенной формой `elif`.

```
>>> hostname = 'NJ'
>>>
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
... elif hostname == 'NJ':
...     print('This hostname is NJ')
...
This hostname is NJ
>>>
```

Tlgm: @it_boooks

Этот оператор очень похож на if, и точно так же условное выражение с его применением завершается символом двоеточия, а соответствующий ему выполняемый блок кода обязательно должен быть смещен вправо. Вероятно, вы заметили, что в коде оператор elif непременно должен быть выровнен по соответствующему оператору if.

А если только NYC и NJ являются правильными и допустимыми именами хостов, но необходимо выполнить блок кода, если обнаружено другое имя хоста? Тогда мы воспользуемся оператором else.

```
>>> hostname = 'DEN_CO'
>>>
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
... elif hostname == 'NJ':
...     print('This hostname is NJ')
... else:
...     print('UNKNOWN HOSTNAME')
...
UNKNOWN HOSTNAME
>>>
```

Применение оператора else ничем не отличается от применения if и elif. Точно так же необходимо двоеточие (:) в конце условного выражения и смещение вправо соответствующего выполняемого блока.

Когда Python обрабатывает условные выражения, выход из блока условной конструкции происходит сразу же после того, как обнаружено выполнение одного из заданных условий. Например, если переменная hostname содержит строку 'NYC', то условие выполняется в первой строке блока, и выполняется код print('This hostname is NYC'), затем происходит выход из условного блока (без выполнения ветвей elif и else).

Ниже приведен пример с ошибкой, возникающей из-за несоответствия выравнивания взаимосвязанных логических операторов. Перед оператором elif вставлены пробелы, которых быть не должно.

```
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
...   elif hostname == 'NJ':
  File "<stdin>", line 3
    elif hostname == 'NJ':
                   ^
IndentationError: unindent does not match any outer indentation level
>>>
```

Еще один пример демонстрирует ошибку из-за отсутствия двоеточия, завершающего строку условного выражения.

```
>>> if hostname == 'NYC'
  File "<stdin>", line 1
    if hostname == 'NYC'
                       ^
SyntaxError: invalid syntax
>>>
```

Tlgm: @it_boooks

Цель этих примеров: показать, что не следует беспокоиться по поводу опечаток при вводе кода, свойственных начинающим. Вы получите вполне понятные и информативные сообщения об ошибках.

Изучение условных выражений будет продолжено в следующих разделах с использованием практических примеров, в том числе и в следующем разделе, представляющем концепцию хранения объектов в некотором другом объекте.

Концепция объекта, содержащего другие объекты

Термин containment обозначает возможность проверки того факта, что некоторый объект содержит (contains) определенный элемент или объект. Особое внимание здесь будет уделено использованию оператора `in`, основой которого являются рассмотренные в предыдущем разделе условные выражения.

Несмотря на то что в этом разделе оператор `in` рассматривается весьма кратко, не стоит недооценивать мощь этой функциональной возможности в языке Python.

Если вновь обратиться к переменной `vendors`, которая использовалась в предыдущих примерах, то как можно проверить наличие в ее содержимом имени конкретного производителя? Один из вариантов решения – цикл по всему списку и сравнение каждого элемента списка с искомым объектом. Это вполне осуществимо, но почему бы просто не воспользоваться оператором `in`.

Применение возможности проверки содержимого не только более удобно для чтения, но еще и упрощает сам процесс проверки, то есть обнаружение искомого объекта.

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> 'arista' in vendors
True
>>>
```

Синтаксис этого примера вполне понятен, а результатом проверки является логическое значение типа `bool`. Следует отметить, что это еще одно выражение, синтаксис которого является характерной особенностью написания кода на языке Python.

Также можно включить конструкцию проверки содержимого в условное выражение.

```
>>> if 'arista' in vendors:
...     print('Arista is deployed.')
...
'Arista is deployed.'
>>>
```

Следующий пример проверяет, является ли заданная строка частью другой строки. Проверка выполняется точно так же, как поиск заданного элемента в списке. В примере сначала показано простейшее логическое выражение, затем это выражение включается в условное выражение.

Tlgm: @it_boooks


```
>>> version = "CSR1000V Software (X86_64_LINUX_IOSD-UNIVERSALK9-M),
Version 16.3.1, RELEASE"
>>>
>>> "16.3.1" in version
True
>>>
>>> if "16.3.1" in version:
...   print("Version is 16.3.1!!")
...
Version is 16.3.1!!
>>>
>>>
```

Выше уже отмечалось, что операция проверки содержимого в сочетании с условными выражениями представляет собой простой, но мощный способ выяснения того факта, что объект или значение содержится в другом объекте. Начинающие часто увлекаются построением длинных и сложных условных выражений, но в действительности им требуется лишь простой и эффективный способ исследования элементов в заданном объекте.

Один из таких способов – использование циклов при работе с составными объектами, такими как списки и словари. Циклы упрощают обработку объектов подобных типов. В следующем разделе рассматриваются основы работы с циклами.

Использование циклов

Наконец-то мы добрались до циклов. Объекты имеют обыкновение увеличиваться в размерах, особенно объекты в реальных задачах, которые намного больше, чем в приводимых выше простых примерах, поэтому циклы крайне необходимы. В качестве примера можно привести списки устройств, IP-адресов, VLAN-подсетей и интерфейсов. Требуются эффективные способы поиска данных и выполнения некоторых операций с каждым элементом в наборе данных. При решении задач такого рода циклы демонстрируют свое важное значение.

Мы будем рассматривать два основных типа циклов – `for` и `while`.

С точки зрения сетевого инженера, стремящегося к автоматизации сетевых устройств и общей сетевой инфраструктуры, практически всегда можно исключить использование цикла for. Разумеется, это зависит исключительно от того, что именно вы делаете, но, вообще говоря, циклы `for` в языке Python настолько хорошо реализованы, что мы решили оставить их описание в этом разделе.

Использование цикла while

Цикл `while` основан на концепции повторяющегося выполнения некоторого блока кода, пока заданное условие истинно. В следующем примере переменной `counter` присваивается значение `1`, после чего пока значение этой переменной меньше `5`, оно выводится на экран, затем значение `counter` увеличивается на 1.

Tlgm: @it_boooks

Синтаксис этой конструкции похож на синтаксис, который использовался в группе условных выражений `if-elif-else`. Выражение с оператором цикла `while` завершается символом двоеточия (:), а блок выполняемого в цикле кода смещается вправо на четыре пробела.

```
>>> counter = 1
>>>
>>> while counter < 5:
...     print(counter)
...     counter += 1
...
1
2
3
4
>>>
```

Для начального уровня изучения приведенного выше примера вполне достаточно, чтобы понять работу цикла `while`, поэтому переходим к рассмотрению цикла `for` в большинстве следующих примеров.

Использование цикла for

Циклы for в языке Python очень хорошо реализованы и обычно используются для циклической или итеративной (iterating) обработки наборов объектов, таких как список, строка или словарь. В других языках программирования для цикла `for` требуется обязательное определение переменной (счетчика) цикла и постепенное изменение значения этой переменной, но в языке Python это совсем не обязательно.

Начнем с изучения циклов, которые нередко называют *for-in* или *for-each*, представляющих наиболее часто используемый тип цикла `for` в языке Python.

Как и в предыдущих разделах, сначала рассмотрим несколько простейших примеров.

Первый пример – вывод каждого объекта, содержащегося в списке. Из следующего примера видно, что синтаксис прост и очень похож на синтаксис условных выражений и цикла `while`. Первое выражение или заголовок цикла `for` должно завершаться символом двоеточия (:), а блок выполняемого кода непременно должен быть смещен вправо.

```
>>> vendors
['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     print('VENDOR: ' + vendor)
...
VENDOR:  arista
VENDOR:  juniper
VENDOR:  big_switch
VENDOR:  cisco
>>>
```

Tlgm: @it_boooks

Как было сказано выше, этот тип цикла for часто называют *for-in* или *for-each*, потому что он позволяет обработать каждый (*each*) элемент в (*in*) заданном объекте.

В приведенном выше примере имя объекта `vendor` выбрано произвольно, его определяет пользователь, и на каждой итерации имени `vendor` соответствует конкретный элемент обрабатываемого объекта. Например, во время первой итерации имени `vendor` соответствует элемент `arista`, во время второй итерации – `juniper` и т. д.

Чтобы убедиться в том, что имя переменной цикла можно выбирать произвольно, заменим его на `network_vendor`.

```
>>> for network_vendor in vendors:
...     print('VENDOR: ' + network_vendor)
...
VENDOR:  arista
VENDOR:  juniper
VENDOR:  big_switch
VENDOR:  cisco
>>>
```

А теперь объединим все, что к настоящему моменту мы узнали об условных выражениях, объектах, содержащих другие объекты, и циклах в одном более сложном примере.

В этом примере определяется новый список производителей `vendors`. Одно из имен представляет известную компанию, но эта компания не является производителем сетевого оборудования. Затем определяется список `approved_vendors`, в котором перечислены действительные производители сетевого оборудования, утвержденные для конкретного клиента. Далее выполняется циклический проход по списку производителей и проверяется, действительно ли рассматриваемый на текущей итерации элемент является именем утвержденного производителя. Если условие не выполняется, то выводится соответствующее сообщение.

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco', 'oreilly']
>>>
>>> approved_vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     if vendor not in approved_vendors:
...         print('NETWORK VENDOR NOT APPROVED: ' + vendor)
...
NETWORK VENDOR NOT APPROVED:  oreilly
>>>
```

Здесь мы видим, что оператор логического отрицания `not` применяется в сочетании с оператором `in`, используя всю мощь этих функциональных возможностей и упрощая понимание (чтение) происходящего.

Рассмотрим еще более сложный пример с обработкой в цикле словаря и одновременным извлечением данных из другого словаря, а также с использованием некоторых встроенных методов, изученных ранее в этой главе.

Tlgm: @it_boooks

Чтобы подготовиться к следующему примеру, создадим словарь, содержащий команды CLI-интерфейса для конфигурирования некоторых функциональных характеристик сетевого устройства:

```
>>> COMMANDS = {
...     'description': 'description {}',
...     'speed': 'speed {}',
...     'duplex': 'duplex {}',
... }
>>>
>>> print(COMMANDS)
{'duplex': 'duplex {}', 'speed': 'speed {}', 'description': 'description {}'}
>>>
```

В этом словаре три элемента (три пары ключ-значение). Ключом каждого элемента является конфигурируемая функциональная характеристика сетевого устройства, а значение представляет начало командной строки, которая будет выполнять конфигурирование соответствующей характеристики. Ключами являются функциональные характеристики description, speed и duplex. В каждое значение словаря включены фигурные скобки ({}), поскольку в примере будет использоваться метод `format()` для подстановки в строки значений переменных.

После создания словаря `COMMANDS` сформируем второй словарь `CONFIG_PARAMS`, который будет использоваться для того, чтобы определить, какие команды будут выполняться, и какие значения будут подставляться в командные строки, выбираемые из словаря `COMMANDS`.

```
>>> CONFIG_PARAMS = {
...     'description': 'auto description by Python',
...     'speed': '10000',
...     'duplex': 'auto'
... }
>>>
```

Теперь воспользуемся циклом `for` для итеративного прохода по словарю `CONFIG_PARAMS` с применением встроенного метода `items()`. На каждой итерации берется ключ из словаря `CONFIG_PARAMS` и используется для извлечения с помощью встроенного метода `get()` соответствующего значения, то есть командной строки из словаря `COMMANDS`. Это возможно благодаря тому, что при создании словарей использовалась одинаковая структура ключей. Командная строка извлекается с фигурными скобками, но к ней сразу же применяется метод `format()` для вставки правильного значения, взятого из соответствующего элемента словаря `CONFIG_PARAMS`.

Ниже приводится окончательный код примера.

```
>>> commands_list = []
>>>
>>> for feature, value in CONFIG_PARAMS.items():
...     command = COMMANDS.get(feature).format(value)
...     commands_list.append(command)
```

Tlgm: @it_boooks


```
...
>>> commands_list.insert(0, 'interface Eth1/1')
>>>
>>> print(commands_list)
['interface Eth1/1', 'duplex auto', 'speed 10000',
  'description auto description by Python']
>>>
```

Разберем этот пример более подробно. Рекомендуем потратить немного времени и собственноручно протестировать данный пример в интерактивном интерпретаторе Python.

В первой строке создается пустой список `commands_list`. В дальнейшем к этому списку будет применяться встроенный метод `append()` для добавления элементов.

Затем встроенный метод `items()` применяется при циклическом проходе по словарю `CONFIG_PARAMS`. В текущей главе уже встречалось весьма краткое описание этого метода, но еще раз напомним, что метод `items()` предоставляет разработчику сетевой среды одновременный доступ к ключу и значению конкретной пары ключ-значение. В рассматриваемом примере выполняется итеративный проход по трем парам ключ-значение: description/auto description by Python, speed/10000, и duplex/auto.

Во время каждой итерации, то есть для каждой пары ключ-значение, с предоставлением доступа через переменные `feature` и `value` извлекается соответствующая команда из словаря `COMMANDS`. Как вы помните, метод `get()` используется для извлечения пары ключ-значение по заданному ключу. В примере ключом является объект `feature`. Возвращаемыми значениями являются `description` {} для `description`, `speed` {} для `speed` и `duplex` {} для `duplex`. Все эти объекты возвращаются как строки, поэтому мы сразу можем применить метод `format()` для вставки значений `value` из словаря `CONFIG_PARAMS`, так как в примерах предыдущих разделов была продемонстрирована возможность совместного применения нескольких методов в одной строке кода.

После подстановки значения полученная команда добавляется в список `commands_list`. Когда список команд сформирован полностью, в начало добавляется элемент `Eth1/1` с помощью метода `insert()`. Эта операция может быть выполнена самой первой.

Если вам полностью понятен данный пример, то можете считать, что достаточно хорошо освоили программирование на языке Python на начальном уровне обучения.

Мы рассмотрели наиболее часто используемые типы циклов `for`, которые позволяют выполнять итеративный проход по спискам и словарям. Теперь перейдем к другим способам формирования и применения цикла `for`.

Использование функции enumerate()

Иногда необходимо отслеживать значение индекса во время обработки объекта в цикле. Эту тему мы рассмотрим очень кратко, так как демонстрируемые здесь примеры похожи на ранее приведенные в предыдущих разделах.

Tlgm: @it_boooks

Функция `enumerate()` используется для нумерации списка и получения значения индекса, поэтому часто оказывается полезной для определения точной позиции заданного элемента.

В следующем примере показано, как применить функцию `enumerate()` в цикле `for`. Отметим, что заголовок цикла `for` выглядит почти так же, как в примере обработки словаря, но вместо метода `items()`, возвращающего ключ и значение, указана функция `enumerate()`, возвращающая индекс, пронумерованный с 0, и объект из списка, соответствующий текущему значению индекса (номеру).

В примере выводится индекс и значение (элемент), чтобы облегчить понимание происходящего.

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for index, each in enumerate(vendors):
...     print(index + ' ' + each)
...
0 arista
1 juniper
2 big_switch
3 cisco
>>>
```

Иногда не требуется выводить все индексы и значения. Может возникнуть необходимость в выводе только одного индексного номера для определенного производителя. Такой вариант показан в следующем примере.

```
>>> for index, each in enumerate(vendors):
...     if each == 'arista':
...         print('arista index is: ' + index)
...
arista index is:  0
>>>
```

К настоящему моменту мы изучили лишь немногие характеристики и возможности языка Python – от типов данных до условных выражений и циклов. Но мы пока еще не рассматривали эффективное многократное использование кода с помощью функций. Это тема следующего раздела.

Использование функций

Заинтересованные читатели этой книги, возможно, знают кое-что о функциях, но даже если это не так, не стоит беспокоиться – этот раздел пополнит ваши знания. Функции (functions) предназначены для устранения избыточности и дублирования кода, то есть позволяют использовать код многократно. Вообще говоря, функции по своей сущности противоположны действиям, которые сетевые инженеры повторяют изо дня в день.

Ежедневно сетевые инженеры конфигурируют виртуальные локальные сети (VLAN) снова и снова. Вероятно, они в некоторой степени гордятся тем, как

Tlgm: @it_boooks

быстро они способны вводить одни и те же команды в командной строке и отправлять их в сетевое устройство или коммутатор раз за разом. Написанный один раз скрипт с функциями устраняет необходимость многократного ввода вручную одних и тех же команд.

Предположим, что нужно создать несколько виртуальных сетей VLAN с использованием некоторого набора коммутаторов. Если это устройства компаний Cisco или Arista, то требуемые команды могут выглядеть следующим образом:

```
vlan 10
  name USERS
vlan 20
  name VOICE
vlan 30
  name WLAN
```

А теперь представьте себе, что необходимо конфигурировать 10, 20 или 50 устройств для создания тех же виртуальных сетей. Вероятнее всего, потребуется вручную ввести показанные выше шесть команд столько раз, сколько устройств имеется в вашей сетевой среде.

Это как раз тот самый случай, когда появляется возможность создать функцию и написать небольшой скрипт. Мы пока еще не рассматривали скрипты как независимые программы, поэтому продолжаем работать в интерактивной оболочке Python shell.

В нашем первом примере начнем с создания простейшей функции `print_vendor()`, затем вернемся к примеру создания виртуальных локальных сетей.

```
>>> def print_vendor(net_vendor):
...     print(net_vendor)
...
>>>
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     print_vendor(vendor)
...
arista
juniper
big_switch
cisco
>>>
```

Здесь `print_vendor()` – функция, созданная и определенная (defined) с использованием ключевого слова `def`. Если необходимо передать переменные (параметры) в функцию, то их имена указываются в круглых скобках, следующих за именем функции. В приведенном примере передается один параметр с именем `vendor`, который принимает функция `print_vendor()` и использует в своем внутреннем коде. Подобно условным выражениям и циклам, строка объявления функции также должна завершаться символом двоеточия (`:`). Внутри

Tlgm: @it_boooks

функции располагается смещенный вправо блок кода, который в данном случае состоит из одной строки – принятый параметр просто выводится на экран.

После создания функция сразу же готова к использованию, даже в среде интерактивного интерпретатора Python.

В первом примере мы обеспечили создание списка `vendors`, затем обработали его в цикле. На каждой итерации цикла в функцию `print_vendor()` передается объект, являющийся строкой с именем производителя оборудования.

Отметим, что переменные имеют различные имена в зависимости от того, где они используются. Таким образом, в функцию извне передается имя переменной `vendor`, но передаваемое значение принимается и используется внутри функции как переменная с именем `net_vendor`. Не существует никаких ограничений или специальных требований на использование переменных с одинаковыми именами вне функции и внутри ее, и если вы выберете такой способ именования, все будет работать.

После того как было подробно описано создание простой функции, вернемся к примеру создания VLAN.

Для автоматизации процедуры создания VLAN создаются две функции.

Первая функция `get_commands()` формирует требуемые команды для передачи их в сетевое устройство. Она принимает два параметра: `vlan` – идентификатор VLAN ID и `name` – имя VLAN.

Вторая функция `push_commands()` передает готовые команды, сформированные функцией `get_commands()`, в устройства, указанные в заданном списке. Эта функция также принимает два параметра: `device` – имя устройства, в которое передается команда, и `commands` – список передаваемых команд. В нашем учебном примере реальная передача команд в устройства не производится, вместо этого команды просто выводятся на экран для имитации их выполнения.

```
>>> def get_commands(vlan, name):
...     commands = []
...     commands.append('vlan ' + vlan)
...     commands.append('name ' + name)
...
...     return commands
...
>>>
>>> def push_commands(device, commands):
...     print('Connecting to device: ' + device)
...     for cmd in commands:
...         print('Sending command: ' + cmd)
>>>
```

Для практического использования этих функций необходим список конфигурируемых устройств и список виртуальных локальных сетей VLAN, передаваемых в устройства.

Список конфигурируемых устройств может выглядеть следующим образом:

```
>>> devices = ['switch1', 'switch2', 'switch3']
>>>
```

Tlgm: @it_boooks

Чтобы получить единый объект, представляющий виртуальные сети VLAN, создается список словарей. В каждом словаре содержится две пары ключ-значение, одна пара для идентификатора VLAN ID, вторая – для имени VLAN.

```
>>> vlans = [{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
{'id': '30', 'name': 'WLAN'}]
>>>
```

Как вы помните, существует несколько способов создания словарей. Здесь можно было бы применить любой из этих способов.

В следующем фрагменте кода показан один вариант использования созданных выше функций. Здесь выполняется проход по списку `vlans` в цикле. Напомним, что каждый элемент в списке `vlans` является словарем. Из каждого элемента, то есть словаря, с помощью встроенного метода `get()` извлекается идентификатор `id` и имя `name`. Далее следуют два вызова функции `print()`, затем вызывается наша первая функция `get_commands()`, и в нее передаются параметры `id` и `name`. Функция возвращает список команд, который присваивается переменной `commands`.

После этого команды создания требуемых виртуальных сетей VLAN выполняются на каждом устройстве в цикле прохода по списку `device`. На каждой итерации цикла вызывается функция `push_commands()`, передающая команды создания всех VLAN в каждое устройство.

Ниже приводится соответствующий код и вывод результатов его работы:

```
>>> for vlan in vlans:
...     id = vlan.get('id')
...     name = vlan.get('name')
...     print('\n')
...     print('CONFIGURING VLAN:' + id)
...     commands = get_commands(id, name)
...     for device in devices:
...         push_commands(device, commands)
...         print('\n')
...
>>>

CONFIGURING VLAN: 10
Connecting to device:  switch1
Sending command:  vlan 10
Sending command:  name USERS

Connecting to device:  switch2
Sending command:  vlan 10
Sending command:  name USERS

Connecting to device:  switch3
Sending command:  vlan 10
Sending command:  name USERS

CONFIGURING VLAN: 20
Connecting to device:  switch1
```

Tlgm: @it_boooks


```
Sending command:  vlan 20
Sending command:  name VOICE

Connecting to device:  switch2
Sending command:  vlan 20
Sending command:  name VOICE

Connecting to device:  switch3
Sending command:  vlan 20
Sending command:  name VOICE

CONFIGURING VLAN: 30
Connecting to device:  switch1
Sending command:  vlan 30
Sending command:  name WLAN

Connecting to device:  switch2
Sending command:  vlan 30
Sending command:  name WLAN

Connecting to device:  switch3
Sending command:  vlan 30
Sending command:  name WLAN
>>>
```



Помните: не для всех функций требуются параметры, и не все функции возвращают значения.

Теперь вы знаете, как создаются и используются функции, понимаете, как они определяются и вызываются с параметрами и без параметров, а также умеете применять функции в циклах.

Следующая тема – чтение и запись данных в файлы средствами языка Python.

Работа с файлами

Главная цель этого раздела – продемонстрировать операции чтения и записи данных в файлы. Здесь рассматриваются только основы, необходимые для начала работы с файлами, но вы можете обратиться к любому учебному пособию по языку Python, чтобы получить более подробную информацию по этой теме.

Чтение данных из файла

Для учебного примера воспользуемся фрагментом файла конфигурации, расположенного в том каталоге, из которого был выполнен вход в интерпретатор Python.

Ниже приводится приблизительное содержание конфигурационного файла *vlans.cfg*:

```
vlan 10
  name USERS
vlan 20
```

Tlgm: @it_boooks


```
  name VOICE
vlan 30
  name WLAN
vlan 40
  name APP
vlan 50
  name WEB
vlan 60
  name DB
```

С помощью всего лишь двух строк команд на языке Python можно открыть (open) и прочитать (read) содержимое этого файла.

```
>>> vlans_file = open('vlans.cfg', 'r')
>>>
>>> vlans_file.read()
'vlan 10\n  name USERS\nvlan 20\n  name VOICE\nvlan 30\n
name WLAN\nvlan 40\n  name APP\nvlan 50\n  name WEB\nvlan 60\n  name DB'
>>>
>>> vlans_file.close()
>>>
```

В этом примере выполняется чтение всего содержимого файла в один объект строкового типа str с помощью метода `read()` для файловых объектов.

В следующем примере считывается содержимое того же файла, но каждая строка сохраняется как элемент списка с помощью метода `readlines()` для файловых объектов.

```
>>> vlans_file = open('vlans.cfg', 'r')
>>>
>>> vlans_file.readlines()
['vlan 10\n', '  name USERS\n', 'vlan 20\n', '  name VOICE\n', 'vlan 30\n',
'  name WLAN\n', 'vlan 40\n', '  name APP\n', 'vlan 50\n', '  name WEB\n',
'vlan 60\n', '  name DB']
>>>
>>> vlans_file.close()
>>>
```

Теперь еще раз откроем тот же файл, сохраним его содержимое как строку, но затем обработаем ее, чтобы записать характеристики VLAN в форме словаря способом, аналогичным обработке объекта `vlans` в примере из раздела, в котором рассматривались функции.

```
>>> vlans_file = open('vlans.cfg', 'r')
>>>
>>> vlans_text = vlans_file.read()
>>>
>>> vlans_list = vlans_text.splitlines()
>>>
>>> vlans_list
['vlan 10', '  name USERS', 'vlan 20', '  name VOICE', 'vlan 30',
'  name WLAN', 'vlan 40', '  name APP', 'vlan 50', '  name WEB',
```

Tlgm: @it_boooks


```
'vlan 60', '  name DB']
>>>
>>> vlans = []
>>> for item in vlans_list:
...     if 'vlan' in item:
...         temp = {}
...         id = item.strip().strip('vlan').strip()
...         temp['id'] = id
...     elif 'name' in item:
...         name = item.strip().strip('name').strip()
...         temp['name'] = name
...         vlans.append(temp)
...
>>>
>>> vlans
[{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
{'id': '30', 'name': 'WLAN'}, {'id': '40', 'name': 'APP'},
{'id': '50', 'name': 'WEB'}, {'id': '60', 'name': 'DB'}]
>>>
>>> vlans_file.close()
>>>
```

В этом примере содержимое файла считывается и сохраняется в строке `vlans_text`. Встроенный метод `splitlines()` для строк используется для создания списка, в котором каждый элемент представляет отдельную строку исходного файла. Новый список называется `vlans_list`, а его длина равна количеству команд, содержащихся в исходном файле.

После создания список последовательно обрабатывается в цикле `for`. Переменная `item` используется для представления каждого элемента списка, обрабатываемого на текущей итерации. На первой итерации переменная `item` содержит строку `'vlan 10'`, на второй итерации – строку `' name users'` и т. д. В теле цикла `for` окончательно формируется список словарей, в котором каждый элемент представляет собой словарь с двумя парами ключ-значение: `id` и `name`. Эта операция выполняется с использованием временного словаря `temp`, в который добавляются обе пары ключ-значение. И только после добавления имени VLAN содержимое временного словаря записывается в итоговый список. Таким образом, временный словарь `temp` реинициализируется только в том случае, когда обнаруживается очередное имя VLAN.

Обратите внимание на использование встроенного метода `strip()`. Этот метод можно использовать не только для удаления пробелов, но и для удаления заданных фрагментов (подстрок) в строковом объекте. Кроме того, здесь используется объединение в цепочку последовательных вызовов нескольких методов в одной команде языка Python.

Например, для значения `' name WEB'` сначала вызывается метод `strip()`, который возвращает строку `'name WEB'`. Затем вызывается метод `strip('name')`, возвращающий строку `' WEB'`, и завершает цепочку вызовов метод `strip()` для удаления всех оставшихся пробелов и получения окончательного имени `'WEB'`.

Tlgm: @it_boooks



Приведенный выше пример не является единственным способом выполнения операции чтения характеристик VLAN. В приведенном примере предполагается наличие идентификатора VLAN ID и имени для каждой виртуальной локальной сети, но обычно это не так. Тем не менее этот способ приведен в качестве иллюстрации некоторых концепций. Словарь `temp` инициализируется только в случае обнаружения подстроки «vlan», а сам словарь `temp` включается в общий список только после добавления значения имени `name` (этот способ не будет работать, если не для каждой сети VLAN существует имя, но в этом случае вполне уместно применить механизм обработки ошибок языка Python, воспользовавшись операторами `try/except`, которые в нашей книге не рассматриваются).

Запись данных в файл

В следующем примере показано, как записать данные в файл.

Здесь используется объект vlan, созданный в примере предыдущего раздела.

```
>>> vlans
[{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
{'id': '30', 'name': 'WLAN'}, {'id': '40', 'name': 'APP'},
{'id': '50', 'name': 'WEB'}, {'id': '60', 'name': 'DB'}]
```

Перед операцией записи всего списка в новый файл создается еще несколько VLAN.

```
>>> add_vlan = {'id': '70', 'name': 'MISC'}
>>> vlans.append(add_vlan)
>>>
>>> add_vlan = {'id': '80', 'name': 'HQ'}
>>> vlans.append(add_vlan)
>>>
>>> print(vlans)
[{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
{'id': '30', 'name': 'WLAN'}, {'id': '40', 'name': 'APP'},
{'id': '50', 'name': 'WEB'}, {'id': '60', 'name': 'DB'},
{'id': '70', 'name': 'MISC'}, {'id': '80', 'name': 'HQ'}]
>>>
```

Теперь в списке vlans содержатся данные о восьми виртуальных сетях VLAN. Запишем их в новый файл, но при этом необходимо соблюдать условия правильного форматирования с помощью добавления требуемого количества пробелов.

Сначала необходимо открыть новый файл. Если файл не существует, как в нашем случае, он будет создан. Эта операция выполняется в первой строке кода, приведенного ниже.

После того как файл открыт, применяется метод `get()` для извлечения значений VLAN из каждого словаря, затем вызывается встроенный файловый метод `write()` для записи данных в файл. После завершения всех операций записи файл закрывается.

```
>>> write_file = open('vlans_new.cfg', 'w')
>>>
```

Tlgm: @it_boooks


```
>>> for vlan in vlans:
...     id = vlan.get('id')
...     name = vlan.get('name')
...     write_file.write('vlan ' + id + '\n')
...     write_file.write('  name ' + name + '\n')
...
>>>
>>> write_file.close()
>>>
```

В приведенном выше коде создан файл *vlans_new.cfg* и сгенерировано следующее содержимое этого файла:

```
$ cat vlans_new.cfg
vlan 10
  name USERS
vlan 20
  name VOICE
vlan 30
  name WLAN
vlan 40
  name APP
vlan 50
  name WEB
vlan 60
  name DB
vlan 70
  name MISC
vlan 80
  name HQ
```

Когда вы начнете работать с файловыми объектами на практике, то можете встретиться с некоторыми непонятными явлениями. Например, если вы забыли закрыть файл в конце скрипта, то вас очень удивит отсутствие данных в файле, хотя вы точно знаете, что они должны быть там.

По умолчанию все, что записывается с помощью метода `write()`, сохраняется в специальном буфере, а реальная запись в файл выполняется только после команды закрытия файла. Но параметр конфигурации, который управляет процедурой записи в файл, можно изменить.

Кроме того, можно воспользоваться ключевым словом (командой) `with` – менеджером контекста – для управления этим процессом.

Ниже приведен краткий пример использования команды `with`. Одним из преимуществ этой команды является то, что она автоматически закрывает файл.

```
>>> with open('vlans_new.cfg', 'w') as write_file:
...     write_file.write('vlan 10\n')
...     write_file.write('  name TEST_VLAN\n')
...
>>>
```

Tlgm: @it_boooks

Когда файл открывается с помощью команды `open('vlans.cfg', 'r')`, очевидно, что в метод передаются два параметра. Первый параметр – имя файла, которое может включать относительный или абсолютный путь к нему. Второй параметр определяет режим (mode), но это необязательный параметр, и если он отсутствует, то по умолчанию назначается режим только для чтения, то есть отсутствие второго параметра равнозначно режиму `r`. Другие возможные режимы: `w` – файл открывается только для записи (если файл с указанным именем уже существует, то его содержимое уничтожается), `a` – файл открывается для добавления данных к уже существующим, `r+` – файл открывается для чтения и записи.

До настоящего момента все примеры в этой главе выполнялись в динамическом интерактивном интерпретаторе Python. Были продемонстрированы мощные возможности интерпретатора для написания и тестирования новых методов, функций или отдельных фрагментов кода. Но при всем богатстве возможностей интерактивной оболочки Python сохраняется необходимость написания программ и скриптов, которые должны работать независимо от интерпретатора. Это тема следующего раздела.

Создание программ на языке Python

Не забывая о том, как мы писали код и выполняли его в интерпретаторе Python shell, в этом разделе мы научимся создавать и выполнять независимые скрипты или программы на языке Python. Здесь будет продемонстрировано, как можно с легкостью написать скрипт буквально за несколько минут, используя лишь те знания, которые были получены при изучении предыдущих разделов.

Для написания кода следующих примеров вы можете выбрать любой текстовый редактор, который наиболее удобен для вас. Можно порекомендовать vi, vim, Sublime Text, Notepad++ или даже полноценную среду разработки (IDE), такую как PyCharm, но окончательное решение остается за вами.

Рассмотрим несколько примеров.

Создание простого скрипта на языке Python

Сначала создается новый файл с расширением *.py*. В терминале Linux можно создать файл командой `touch net_script.py`, затем открыть этот файл в каком-либо текстовом редакторе. Разумеется, этот файл совершенно пуст.

Наш первый скрипт будет просто выводить текст в терминале. Добавьте следующие пять строк текста в файл *net_script.py*, чтобы создать простейший скрипт.

```
#!/usr/bin/env python

if __name__ == "__main__":
    print('^' * 30)
    print('HELLO NETWORK AUTOMATION!!!!!')
    print('^' * 30)
```

Tlgm: @it_boooks

После создания скрипта (не забудьте сохранить и закрыть файл) выполним его.

Для выполнения скрипта на языке Python в терминале Linux используется команда `python`. К этой команде нужно добавить имя файла скрипта, как показано ниже.

```
$ python net_script.py
^^^^^^^^^^^^^^^^^^^^^^^^^^^^
HELLO NETWORK AUTOMATION!!!!!
^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Вы создали скрипт на языке Python, который действительно работает. Обратите внимание: все, что располагается ниже строки `if __name__ == "__main__":`, – это практически то же самое, что выполнялось в интерактивном режиме в интерпретаторе Python.

Теперь более подробно рассмотрим две особенные строки команд, которые не являются обязательными, но настоятельно рекомендуемыми при написании скриптов на языке Python. Первая особенность – shebang. Возможно, вы помните, что первое упоминание о shebang встречалось в главе 3.

Что такое shebang

Строка с комбинацией символов shebang (`#!`)[1] является самой первой строкой в приведенном выше примере скрипта: `#!/usr/bin/env python`. Это особенная строка в программах на языке Python.

Это единственная строка кода, в которой первый символ # не обозначает начало комментария. Комментарии будут рассматриваться немного позже, но сейчас мы просто отметим, что символ # часто используется для создания комментариев в коде Python. Комбинация shebang является исключением, кроме того, всегда должна входить в самую первую строку Python-программы.



Статические анализаторы кода (linters) Python, используемые для проверки правильности исходного кода, способны также обрабатывать текст в комментариях, начинающихся с символа #.

Строка, начинающаяся с комбинации символов shebang, сообщает системе, что для выполнения программы используется интерпретатор Python. Разумеется, при этом предполагается, что для файла программы установлены соответствующие права доступа (то есть файл программы должен быть выполняемым). Если shebang-строка не включена в скрипт, то для его выполнения необходимо использовать команду `python`. Тем не менее эту команду мы используем во всех наших примерах.

---

[1] В русскоязычной технической литературе предлагается лишь не вполне благозвучная калька «шебанг» (https://ru.wikipedia.org/wiki/Шебанг_(Unix)). Не лучше ли пользоваться оригиналом, тем более что этот термин встречается не так уж часто. – *Прим. перев.*

Tlgm: @it_boooks

Например, имеется следующий скрипт:

```
if __name__ == "__main__":
    print('Hello Network Automation!')
```

Если этот скрипт содержится в файле *hello.py*, то его можно было бы выполнить командой `python hello.py`. Но команда `./hello.py` не будет выполнена. Чтобы обеспечить правильное выполнение команды `./hello.py`, необходимо добавить shebang-строку в начало файла программы, потому что только так система узнает, как нужно выполнять этот скрипт.

В наших примерах shebang-строка `#!/usr/bin/env python` по умолчанию означает использование Python 2.7 в системе, в которой выполнялись примеры для этой книги. Возможно, у вас установлено несколько версий Python, поэтому для применения другой версии потребуется изменить эту строку, например `#!/usr/bin/env python3` для Python 3.



Также заслуживает внимания тот факт, что shebang-строка `/usr/bin/env python` позволяет изменить текущую системную среду (environment), поэтому нет необходимости каждый раз изменять скрипт или создавать отдельный скрипт только для того, чтобы протестировать его в другой версии Python. Можно воспользоваться командой `which python` для проверки места расположения и версии языка Python, используемой в вашей системе. Например, в системе, используемой при написании данной книги, по умолчанию вызывается версия Python 2.7.6:

```
$ which python
/usr/bin/python
$
$ /usr/bin/python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Теперь рассмотрим более внимательно строку `if __name__ == "__main__":`. Глядя на расположение кавычек, можно понять, что `__name__` – это переменная, а `"__main__"` – строка. Когда файл с кодом на языке Python выполняется как независимый скрипт, переменной `__name__` автоматически присваивается значение `"__main__"`. Таким образом, при выполнении команды `python <script>.py` выполняются все команды, расположенные под строкой `if __name__ == "__main__"`.

Здесь может возникнуть вопрос: в каких случаях значение `__name__` не равно `"__main__"`? Ответ на этот вопрос обсуждается в одном из следующих разделов – «Работа с модулями языка Python», но краткий ответ таков: в тех случаях, когда в Python-файлах содержатся определенные объекты для импортирования, но нет необходимости использовать эти файлы как независимые программы.

Теперь нам известен смысл комбинации символов shebang и строки `if __name__ == "__main__":`, и можно продолжить изучение независимых скриптов на языке Python.

Tlgm: @it_boooks

Перемещение кода из интерпретатора Python в независимый скрипт

В следующем примере используется код, который рассматривался в разделе о функциях. Это сделано для того, чтобы продемонстрировать, насколько легко переместить код, написанный и протестированный в интерпретаторе Python, в файл для создания независимого скрипта.

Код скрипта содержится в файле с именем *push.py*.

```
#!/usr/bin/env python

def get_commands(vlan, name):
    commands = []
    commands.append('vlan ' + vlan)
    commands.append('name ' + name)
    return commands

def push_commands(device, commands):
    print('Connecting to device: ' + device)
    for cmd in commands:
        print('Sending command: ' + cmd)

if __name__ == "__main__":

    devices = ['switch1', 'switch2', 'switch3']

    vlans = [{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
             {'id': '30', 'name': 'WLAN'}]

    for vlan in vlans:
        vid = vlan.get('id')
        name = vlan.get('name')
        print('\n')
        print('CONFIGURING VLAN:' + vid)
        commands = get_commands(vid, name)
        for device in devices:
            push_commands(device, commands)
            print('\n')
```

Этот скрипт можно выполнить с помощью команды `python push.py`.

Результат работы приведенного выше скрипта полностью совпадает с результатом, который был получен при выполнении кода в интерпретаторе Python.

Если бы нужно было создать несколько скриптов, вносящих разнообразные изменения в файлы конфигурации сети, то можно с большой вероятностью предположить, что функция `push_commands()` потребовалась бы почти во всех таких скриптах. Конечно, можно просто скопировать и вставить эту функцию во все скрипты. Но это не самое оптимальное решение, поскольку при необходимости исправления ошибок и внесения изменений в эту функцию придется исправлять все скрипты, которые ее содержат.

Функции позволяют многократно использовать код в одном скрипте, но подобная возможность существует и для многократного использования кода

Tlgm: @it_boooks

в различных скриптах и программах. Это делается с помощью модулей, которые мы будем рассматривать в следующем разделе, продолжая улучшать код нашего учебного примера.

Работа с модулями языка Python

Мы будем продолжать работу с файлом *push.py*, созданным в предыдущем разделе, чтобы наглядно продемонстрировать работу с модулями языка Python. Модуль (module) можно представить как специальный тип Python-файла, в котором хранится информация (например, объекты Python) и который могут использовать другие программы на языке Python, но сам этот файл не является независимо выполняемой программой или скриптом.

Для продолжения работы над примером необходимо временно вернуться в интерпретатор Python, запустив его в том каталоге, где находится файл *push.py*.

Предположим, что нужно сгенерировать новый список команд для передачи в новый список сетевых устройств. Нам известно, что функция `push_commands()` находится в другом файле, в котором уже определена логика передачи списка команд в заданное устройство. Чтобы не создавать заново точно такую же функцию в новой программе (или в интерпретаторе), можно повторно использовать функцию `push_commands()` прямо из файла *push.py*. Рассмотрим, как это делается.

В интерактивной оболочке Python shell нужно ввести команду `import push` и нажать клавишу **Enter**. Эта команда импортирует все объекты из файла *push.py*.

```
>>> import push
>>>
```

Импортированные объекты можно просмотреть с помощью следующей команды:

```
>>> dir(push)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'get_commands', 'push_
commands']
>>>
```

Как и для стандартных типов данных языка Python, для `push` имеются методы, имена которых начинаются и заканчиваются символами подчеркивания, но, кроме них, на себя обращают внимание два метода – `get_commands` и `push_commands`, которые в действительности являются функциями из файла *push.py*.

Как вы помните, функция `push_commands()` принимает два параметра: первый – имя устройства, второй – список команд. Теперь можно воспользоваться функцией `push_commands()` прямо в интерпретаторе.

```
>>> device = 'router1'
>>> commands = ['interface Eth1/1', 'shutdown']
>>>
```

Tlgm: @it_boooks


```
>>> push.push_commands(device, commands)
Connecting to device:  router1
Sending command:  interface Eth1/1
Sending command:  shutdown
>>>
```

Сначала были созданы две новые переменные (`device` и `commands`), которые использовались как параметры, передаваемые в функцию `push_command()`.

Затем вызывается функция `push_commands()` как объект модуля `push` с параметрами `device` и `commands`.

При импортировании нескольких модулей существует вероятность совпадения имен функций, тем не менее показанный выше способ с использованием команды `import push` остается правильным вариантом. Он также позволяет с легкостью узнать, где (в каком модуле) имеется требуемая функция. Но существуют и другие способы импортирования объектов.

Одним из таких способов является команда `from import`. В нашем примере она выглядит так: `from push import push_commands`. Отметим, что в следующем фрагменте кода можно пользоваться функцией `push_commands()` без ссылки на модуль `push`.

```
>>> from push import push_commands
>>>
>>> device = 'router1'
>>> commands = ['interface Eth1/1', 'shutdown']
>>>
>>> push_commands(device, commands)
Connecting to device:  router1
Sending command:  interface Eth1/1
Sending command:  shutdown
>>>
```



Рекомендуется применять команды `import` в тех случаях, когда это действительно необходимо, и импортировать только те объекты, которые используются в коде. Не рекомендуется использовать в этих командах символы-шаблоны, как, например, `from push import *`. Подобные команды загружают все объекты из указанного модуля, создавая чрезмерную перегруженность пространства имен и повышая потенциальную вероятность конфликтов с именами объектов, которые определены в текущем коде. Кроме того, это затрудняет поиск ошибок и устранение проблем, так как сложно выяснить, был ли проблемный объект определен в текущем коде или импортирован из внешнего модуля.

Существует еще одна возможность – переименование импортируемого объекта с помощью команды `from import as`. Если имя объекта по каким-либо причинам не подходит для использования или кажется слишком длинным, можно переименовать его при импортировании, например:

```
>>> from push import push_commands as pc
>>>
```

Обратите внимание на то, насколько просто можно переименовать объект и сделать его имя более коротким и удобным для частого применения.

Tlgm: @it_boooks

Теперь воспользуемся этой же командой в нашем примере.

```
>>> from push import push_commands as pc
>>>
>>> device = 'router1'
>>> commands = ['interface Eth1/1', 'shutdown']
>>>
>>> pc(device, commands)
Connecting to device:  router1
Sending command:  interface Eth1/1
Sending command:  shutdown
>>>
```



В приведенных выше примерах вход в динамический интерпретатор Python выполнялся из каталога, в котором находится файл модуля push.py. Чтобы использовать этот модуль или любой другой новый модуль, находясь в произвольной локации файловой системы, необходимо разместить требуемый модуль в каталоге, который определен в переменной среды PYTHONPATH. Это системная переменная среды в ОС Linux, определяющая все каталоги, в которых система будет искать модули и программы на языке Python.

Теперь вы знаете не только о том, как создать скрипт, но также как создать Python-модуль с функциями (и другими объектами) и как многократно использовать эти объекты в других скриптах и программах.

Передача аргументов в скрипт

В следующих двух разделах мы продолжим изучение создания скриптов на языке Python и практического применения Python-модулей. Сначала рассмотрим один из модулей, являющихся частью стандартной библиотеки Python (то есть входящих в дистрибутивный комплект Python по умолчанию). Этот модуль поддерживает передачу аргументов из командной строки в скрипт на языке Python и называется sys. Мы будем использовать атрибут (или переменную) argv из модуля sys.

Сначала рассмотрим простой скрипт, сохраненный в файле *send_command.py*, который выполняет только одну команду print.

```
#!/usr/bin/env python

import sys

if __name__ == "__main__":
    print(sys.argv)
```

Выполним этот скрипт с передачей в него нескольких аргументов, имитирующих данные, которые необходимы для регистрации на заданном устройстве и выполнения команды show.

```
ntc@ntc:~$ python send-command.py username password 10.1.10.10 "show version"
['send-command.py', 'username', 'password', '10.1.10.10', 'show version']
ntc@ntc:~$
```

Tlgm: @it_boooks

Здесь мы видим, что sys.argv представляет собой список. В действительности это просто список строк, которые передаются в скрипт из командной строки терминала Linux, то есть стандартный список языка Python, элементами которого являются переданные аргументы. Можно догадаться, что произошло на самом деле: интерпретатор Python выполнил операцию разделения (str.split(" ")) строки send-command.py username password 10.1.10.10 "show version" и создал список из пяти элементов.

Отметим также, что при использовании sys.argv первым элементом списка всегда является имя самого выполняемого скрипта.

Кроме того, можно присваивать значение sys.argv произвольной переменной для облегчения работы с переданными параметрами. В любом случае, элементы списка можно извлекать с помощью соответствующих индексных значений, как показано ниже:

```
#!/usr/bin/env python

import sys

if __name__ == "__main__":
    args = sys.argv
    print("Username:  " + args[0])
    print("Password:  " + args[1])
    print("Device IP: " + args[2])
    print("Command:   " + args[3])
```

Результат выполнения этого скрипта:

```
ntc@ntc:~$ python send-command.py username password 10.1.10.10 "show version"
Username:   send-command.py
Password:   username
Device IP:  password
Command:    10.1.10.10
ntc@ntc:~$
```

Данный скрипт можно расширять и дополнять, чтобы выполнить более осмысленные задачи в процессе дальнейшего чтения книги. Например, после изучения главы 7 вы сможете передавать параметры похожим образом в скрипт, который действительно устанавливает соединение с устройством, используя для этого API, выполняющий команду show (или аналогичную ей команду).



При использовании sys.argv необходимо обеспечить обработку ошибок (как минимум, проверку длины списка параметров). Кроме того, пользователь скрипта обязательно должен знать точный порядок, в котором происходит передача элементов (параметров). Для более сложной обработки параметров следует воспользоваться модулем argparse, который предлагает весьма удобный и понятный для пользователей способ передачи параметров с флагами (flags) и встроенное меню помощи. Но этот модуль не рассматривается в данной книге.

Tlgm: @it_boooks

Использование pip для установки пакетов языка Python

Когда вы начнете применять Python на практике, вероятнее всего, вам потребуется установка дополнительного программного обеспечения, написанного на языке Python. Например, может возникнуть необходимость тестирования автоматизации сетевых устройств с помощью netmiko, широко распространенного в программной среде Python SSH-клиента, который мы рассмотрим в главе 7. Чаще всего для распространения программного обеспечения Python, в том числе и netmiko, используется Python Package Index или PyPI (произносится как «пай-пай» (pie-pie)). Также можно просматривать программы и модули и выполнять поиск непосредственно в репозитории PyPI на сайте https://pypi.python.org/pypi.

Для любого программного обеспечения, размещенного в репозитории PyPI, например netmiko, можно воспользоваться специальной программой `pip`, чтобы установить требуемое ПО на своем компьютере прямо из репозитория PyPI. Программа `pip` – это инструмент установки, который по умолчанию переходит в репозиторий PyPI, загружает (скачивает) необходимое ПО и устанавливает его на ваш компьютер.

В ОС Linux команда установки модуля `netmiko` выполняется в одной строке:

```
ntc@ntc:~$ sudo pip install netmiko
# вывод пропущен
```

При этом модуль `netmiko` устанавливается в один из каталогов, включенных в системный путь поиска (который может быть различным в зависимости от используемой ОС).

По умолчанию всегда устанавливается самая последняя и самая стабильная версия заданного пакета Python. Но иногда может потребоваться установка конкретной (более старой) версии. Это удобно, когда вы не желаете автоматически установить новую версию ПО без ее предварительного тестирования. Такой способ называется закреплением, или фиксацией, версии (pinning). Вы можете закрепить (pin) конкретную версию для установки. В следующем примере показано, как применяется закрепление для установки пакета `netmiko` версии 1.4.2.

```
ntc@ntc:~$ sudo pip install netmiko==1.4.2
# вывод пропущен
```

Кроме того, можно пользоваться программой `pip` для обновления версий ПО. Например, если установлена версия 1.4.2 пакета `netmiko`, то можно обновить этот пакет до самой свежей версии с помощью флага `--upgrade` или `-U`, указанного в командной строке.

```
ntc@ntc:~$ sudo pip install netmiko --upgrade
# вывод пропущен
```

Tlgm: @it_boooks

Достаточно часто возникает необходимость установки пакетов Python из исходных кодов (source). Это означает получение самой свежей версии, например, из репозитория GitHub, системы управления версиями, которую мы будем подробно рассматривать в главе 8. Возможно, программный пакет на GitHub содержит исправления ошибок, критических для вашей работы, но исправленная версия еще не опубликована в репозитории PyPI. В этом случае обычно выполняется команда `git clone`, которая также будет более подробно описана в главе 8.

При выполнении операции клонирования проекта на языке Python из GitHub весьма вероятно, что вы обнаружите в корневом каталоге проекта два файла: *requirements.txt* и *setup.py*. Эти файлы используются для установки пакета Python из исходных кодов. В первом файле перечислен список требований для обеспечения корректного запуска программы. Ниже примеден пример текущих требований из файла *requirements.txt* в проекте `netmiko`:

```
paramiko>=1.13.0
scp>=0.10.0
pyyaml
```

Здесь мы видим, что у пакета `netmiko` имеются три зависимости, обычно обозначаемые как deps (от dependencies). Пакеты, от которых зависит нужный вам пакет, также можно установить вместе с этим пакетом из репозитория PyPI, используя ту же программу установки `pip`.

```
ntc@ntc:~$ sudo pip install -r requirements.text
# вывод пропущен
```

Чтобы правильно установить полноценный пакет `netmiko` (из исходных кодов) с учетом требований, можно также выполнить файл *setup.py*, расположенный в корневом каталоге проекта, после завершения операции клонирования с GitHub.

```
ntc@ntc:~$ sudo python setup.py install
# вывод пропущен
```

По умолчанию установка ПО с помощью файла *setup.py* производится в один из каталогов, входящих в системный путь поиска. Если у вас есть желание внести свой вклад в такой проект на GitHub и активно участвовать в разработке, то можно также установить приложение непосредственно из той локации, где расположены файлы (каталог, из которого вы клонировали проект).

```
ntc@ntc:~$ sudo python setup.py develop
# вывод пропущен
```

Установка выполняется так, как если бы файлы находились в локальном каталоге, из которого запускается пакет `netmiko` в нашем случае. В противном случае, если применяется ключ `install` при запуске файла *setup.py*, необходимо изменить соответствующим образом системный путь поиска для локальной установки `netmiko` (еще один пример устранения проблем при установке).

Tlgm: @it_boooks

Советы, приемы и дополнительная информация по использованию языка Python

Глава завершается разделом, содержащим различные советы, особые приемы программирования и дополнительную информацию по практическому применению языка программирования Python. Все описанное здесь полезно знать при работе с языком Python – некоторые способы вполне доступны даже начинающим, некоторые способы являются более изощренными, но нам хотелось, чтобы вы действительно хорошо подготовились к более глубокому изучению языка Python на основе этой главы, поэтому включили в данный раздел как можно больше вспомогательной информации.

Советы, приемы программирования и дополнительная информация приводятся здесь без соблюдения какого-либо порядка важности или значимости:

- иногда требуется доступ к некоторым частям строки или элементов в списке. Например, нужно извлечь только первый символ или первый элемент. Индексное значение `0` можно использовать не только для строк (этот прием ранее не рассматривался), но и для списков. Если существует переменная `router`, содержащая значение `'DEVICE'`, то `router[0]` возвращает `'D'`. Этот прием можно применять и к спискам (он рассматривался в соответствующем разделе). Но как получить доступ к самому последнему элементу строки или списка? Напомним, что для этого можно воспользоваться индексным значением `-1`. Тогда `router[-1]` возвращает `'E'`. Точно так же можно получить самый последний элемент списка;
- основываясь на предыдущем примере, можно расширить приведенную в нем нотацию для получения нескольких первых или нескольких последних символов строки или элементов списка:

```
>>> hostname = 'DEVICE_12345'
>>>
>>> hostname[4:]
'CE_12345'
>>>
>>> hostname[:-2]
'DEVICE_123'
>>>
```

  Этот прием может стать весьма мощным инструментом синтаксического разбора (парсинга) содержимого объектов разнообразных типов;
- можно преобразовать (или выполнить приведение типа – cast) целое число в строку с помощью функции `str(10)`. Также можно выполнить обратную операцию, то есть преобразование строки в целое число с помощью функции `int('10')`;
- мы пользовались функцией `dir()` только для изучения встроенных методов, при этом также упоминались функции `type()` и `help()`. Ниже показаны полезные практические способы применения всех трех функций вместе:

Tlgm: @it_boooks

- проверка типа данных с помощью функции `type()`;
- проверка доступных встроенных методов для объекта с помощью функции `dir()`;
- после того как вы узнали, какой метод нужно использовать, можно получить информацию о том, как следует его использовать, с помощью функции `help()`.

Ниже приводится пример применения описанного выше способа:

```
>>> hostname = 'router1'
>>>
>>> type(hostname)
<type 'str'>
>>>
>>> dir(hostname)
>>> # вывод пропущен; здесь выводятся все методы, имена которых содержат подстроку
"upper"
>>>
>>> help(hostname.upper)  # вывод пропущен
>>>
```

- если необходимо проверить тип переменной в коде Python, то можно воспользоваться механизмом обработки ошибок (`try/except`), но если требуется точно узнать тип конкретного объекта, то лучше всего это сделать с помощью функции `isinstance()`. Если передаваемый в функцию объект принадлежит к типу, наименование которого также передается в функцию как второй аргумент, то функция возвращает значение `True`:

```
>>> hostname = ''
>>> devices = []
>>> if isinstance(devices, list):
...     print('devices is a list')
...
devices is a list
>>>
>>> if isinstance(hostname, str):
...     print('hostname is a string')
...
hostname is a string
>>>
```

- в предыдущих разделах вы узнали, как использовать интерактивный интерпретатор Python и как создавать скрипты на языке Python. Программная среда Python предлагает для командной строки запуска скрипта флаг `-i`, который вместо обычного выполнения и завершения скрипта позволяет войти в интерпретатор и предоставляет доступ ко всем объектам, содержащимся в скрипте. Это отличный инструмент тестирования.

Пример тестируемого файла *test.py*:

```
if __name__ == "__main__":
    devices = ['r1', 'r2', 'r3']

    hostname = 'router5'
```

Tlgm: @it_boooks

Посмотрим, что происходит, если запустить этот скрипт с флагом `-i`.

```
$ python -i test.py
>>>
>>> print(devices)
['r1', 'r2', 'r3']
>>>
>>> print(hostname)
router5
>>>
```

Отметим, что при выполнении сразу происходит вход в интерпретатор Python shell и предоставляется доступ ко всем объектам скрипта. Весьма удобно, не правда ли;

- если объекты не нулевые (или не пустые), то им соответствует логическое значение `True`. Если объекты нулевые (или пустые), то им соответствует значение `False`. Ниже приводится несколько примеров:

```
>>> devices = []
>>> if not devices:
...     print('devices is empty')
...
devices is empty
>>>
>>> hostname = 'something'
>>>
>>> if hostname:
...    print('hostname is not null')
...
hostname is not null
>>>
```

- в разделе о строках мы рассматривали объединение строк с использованием знака плюс (+), а также изучали применение простого и понятного метода формирования строк `format()`. Существует другой способ формирования строк с использованием специального символа `%`. Ниже приводится пример вставки строк (`s`) в другую строку с помощью этого способа:

```
>>> hostname = 'r5'
>>>
>>> interface = 'Eth1/1'
>>>
>>> test  = 'Device %s has one interface: %s ' % (hostname, interface)
>>>
>>> print(test)
Device r5 has one interface: Eth1/1
>>>
```

- комментарии не рассматривались, но упоминалось, что символ # (хештег, знак номера или знак фунта) используется для создания комментариев в строке кода.

Tlgm: @it_boooks


```
def get_commands(vlan, name):
    commands = []
    # building list of commands to configure a vlan
    commands.append('vlan ' + vlan)
    commands.append('name ' + name)  # appending name
    return commands
```

- строка создания документации (docstring) обычно добавляется в функции, методы и классы. Эта строка кратко описывает, что делает объект. Для ее создания используются три двойные кавычки (""") (в начале и в конце). Обычно docstring ограничивается одной строкой.

```
def get_commands(vlan, name):
    """Принимает команды для конфигурирования VLAN.
    """
    commands = []
    commands.append('vlan ' + vlan)
    commands.append('name ' + name)
    return commands
```

Вы уже знаете, как импортировать модуль, например *push.py*. Сейчас мы еще раз импортируем его, чтобы посмотреть, что происходит, когда запрашивается помощь по функции `get_commands()`, для которой теперь сформирована строка документации docstring.

```
>>> import push
>>>
>>> help(push.get_commands)
```

Справочная информация о функции get_commands из модуля push:

```
get_commands(vlan, name)
    Принимает команды для конфигурирования VLAN.
(END)
>>>
```

Функция `help()` позволяет увидеть все созданные строки документации docstrings. Кроме того, выводится информация о передаваемых параметрах и о типе возвращаемых данных, если эта информация была документирована надлежащим образом.
Для этого необходимо добавить значения `Args` и `Returns` в строку документации docstring.

```
def get_commands(vlan, name):
    """Принимает команды для конфигурирования VLAN.
    Args:
        vlan (int): vlan id (идентификатор)
        name (str): имя виртуальной сети vlan
    Returns:
        Возвращается список команд.
    """

    commands = []
```

Tlgm: @it_boooks


```
    commands.append('vlan ' + vlan)
    commands.append('name ' + name)
    return commands
```

Теперь по команде help() для пользователей этой функции выводится гораздо больший объем информации о том, как нужно ее использовать.

```
>>> import push
>>>
>>> help(push.get_commands)
```

Справочная информация о функции get_commands из модуля push:

```
get_commands(vlan, name)
    Принимает команды для конфигурирования VLAN.

    Args:
        vlan (int): vlan id (идентификатор)
        name (str): имя виртуальной сети vlan

    Returns:
        Возвращается список команд.
(END)
```

- создание собственных классов не рассматривалось в этой главе, потому что классы – это обширная, более сложная тема, но мы приведем здесь краткое введение в основы практического применения классов, поскольку они используются в следующих главах.
  Рассмотрим пример, в котором не только используется, но также импортируется класс, который является частью пакета Python (еще одна новая концепция). Отметим, что это учебный, обобщенный пример. Этот класс не используется в настоящем пакете Python.

```
>>> from vendors.cisco.device import Device
>>>
>>> switch = Device(ip='10.1.1.1', username='cisco', password='cisco')
>>>
>>> switch.show('show version')
# вывод пропущен
```

  В этом примере сначала импортируется класс Device из модуля *device.py*, который является частью пакета Python с именем *vendors* (это всего лишь каталог). Процедура импортирования очень похожа на импортирование, показанное в разделе «Работа с модулями языка Python» ранее в этой главе, а пакет Python представляет собой набор модулей, хранящихся в различных подкаталогах.
  Если сравнивать код этого примера с процедурой импортирования функции push_commands() из раздела «Работа с модулями языка Python», то можно заметить различие. Функция использовалась немедленно, сразу после импортирования, а класс необходимо инициализировать.
  Инициализация класса выполняется следующей командой:

Tlgm: @it_boooks


```
>>> switch = Device(ip='10.1.1.1', username='cisco', password='cisco')
>>>
```

Передаваемые аргументы используются для создания экземпляра класса Device. Если имеется несколько устройств, то можно написать код, подобный следующему:

```
>>> switch1 = Device(ip='10.1.1.1', username='cisco', password='cisco')
>>> switch2 = Device(ip='10.1.1.2', username='cisco', password='cisco')
>>> switch3 = Device(ip='10.1.1.3', username='cisco', password='cisco')
>>>
```

В этом случае каждая переменная является отдельным экземпляром класса Device.



При инициализации класса не всегда используются передаваемые параметры. Все классы отличаются друг от друга, но если параметры применяются, то они передаются в так называемый конструктор класса. В языке Python конструктором является метод с именем `__init__`. Класс без конструктора инициализируется следующим образом: `demo = FakeClass()`. После инициализации к методам созданного экземпляра класса можно обращаться так: `demo.method()`.

После создания и инициализации объекта класса можно использовать его методы (methods). Это очень похоже на использование встроенных методов для типов данных, которое рассматривалось в предыдущих разделах главы. Синтаксис обращения к методу: *`class_object.method()`*.
В следующем примере демонстрируется обращение к методу show(). В реальной сетевой среде метод возвращает данные из соответствующего сетевого устройства.



Еще раз обратим ваше внимание на то, что использование методов объекта какого-либо класса очень похоже на использование встроенных методов различных типов данных, таких как строка, список или словарь. Несмотря на то что создание классов – это отдельная, более сложная тема, которая здесь не рассматривается, вы должны хорошо понимать, как использовать классы на практике.

Если выполнить метод show() для объектов switch2 и switch3, то вполне ожидаемо будут получены корректные данные, возвращаемые из соответствующих устройств, потому что каждый объект является отдельным экземпляром класса Device.
Ниже приведен небольшой пример, в котором показано создание двух объектов класса Device, затем использование этих объектов для получения вывода команды show hostname, выполненной на каждом устройстве. Применяется вспомогательная библиотека, которая по умолчанию позволяет получить возвращаемые данные в формате XML, но при необходимости можно без затруднений изменить формат на JSON.

```
>>> switch1 = Device(ip='nycsw01', username='cisco', password='cisco')
>>> switch2 = Device(ip='nycsw02', username='cisco', password='cisco')
```

Tlgm: @it_boooks


```
>>>
>>> switches = [switch1, switch2]
>>> for switch in switches:
...     response = switch.show('show hostname')[1]
...     print(response)
...
# вывод пропущен
>>>
```

Резюме

В этой главе представлено краткое введение в язык программирования Python для сетевых инженеров. Рассматривались основные концепции, такие как типы данных, условные выражения, циклы, функции, работа с файлами, а также создание модулей Python, которые позволяют многократно использовать один и тот же код в различных программах и скриптах, написанных на языке Python. В конце главы в специальном разделе приведен ряд советов и приемов программирования, а также другая дополнительная информация. Этот раздел следует использовать как краткое справочное руководство при дальнейшем изучении языка Python в контексте автоматизации сетей.

В главе 5 будут рассматриваться разнообразные форматы данных, такие как YAML, JSON и XML. Процесс изучения этих форматов основан на материале, изложенном в текущей главе. Например, то, что вы знаете об использовании модулей Python, поможет упростить работу с этими типами данных. Кроме того, будет показана прямая связь между форматами YAML, JSON и словарями языка Python.

Tlgm: @it_boooks

Глава 5

Форматы и модели данных

Если вы обладаете хотя бы небольшим опытом работы с прикладными программными интерфейсами (API), то, вероятно, встречались с такими терминами, как JSON, XML или YAML. Возможно, вам знакомы термины XSD и YANG. Вместе с этими наименованиями вы также могли услышать термин «язык разметки» (markup language). Но что означают эти термины, и как они связаны с работой в сетевой среде и с автоматизацией сети?

Чтобы обмениваться данными, маршрутизаторам и коммутаторам требуются стандартные протоколы взаимодействия. Точно так же приложениям необходима возможность согласования определенного типа синтаксиса, для того чтобы обмениваться данными друг с другом. С этой целью приложения могут использовать стандартные форматы данных (data formats), такие как JSON и XML (а также некоторые другие форматы). Но приложениям требуется согласование не только по форматированию данных, а еще и соглашение о том, как структурированы данные. Модели данных (data models) определяют структуру данных, хранящихся в конкретном формате.

В этой главе будут рассматриваться некоторые форматы данных, которые наиболее широко используются сетевыми API и инструментальными средствами автоматизации. Будет показано, как разработчик может применять эти инструменты в сетевой среде для выполнения своих задач. Также будут кратко описаны модели данных и их роль в автоматизации сети.

Введение в форматы данных

Программист обычно использует обширный набор разнообразных инструментальных средств для хранения и обработки данных в создаваемых программах. Это могут быть простые переменные (отдельные значения), массивы (группы значений), хеши (пары ключ-значение) или даже специализированные объекты, созданные средствами используемого языка программирования.

Все эти средства абсолютно стандартны в пределах создаваемого программного обеспечения. Но иногда требуется более абстрактный, переносимый меж-

Tlgm: @it_boooks

ду различными приложениями формат данных. Например, пользователю, не знакомому с программированием, необходимо передавать данные из одной программы в другую. Часто программам требуется подобный обмен данными между собой, при этом программы написаны на разных языках. Приходится организовывать нечто похожее на привычную схему «клиент-сервер». Например, многие независимо создаваемые пользовательские интерфейсы (user interfaces – UI) применяются для взаимодействия с провайдерами открытых облачных (cloud) сред. Такая возможность существует (если рассматривать ее в упрощенной форме) благодаря стандартным форматам данных. Суть всего сказанного выше заключается в том, что нам необходим стандартный формат данных, чтобы позволить многочисленным разнообразным программам обмениваться информацией друг с другом и чтобы создать интерфейс для взаимодействия человека с этими программами.

Существует несколько вариантов решения этой задачи. В контексте форматов данных мы будем обсуждать текстовое представление данных, которые внутри программ представлены в виде специализированных конструкций, размещаемых в памяти. Все форматы данных, рассматриваемые в этой главе, имеют мощную поддержку в многочисленных языках программирования и операционных системах. В действительности многие языки программирования, в том числе Python, основы которого вы изучали в главе 4, обладают встроенными инструментальными средствами, существенно упрощающими импорт и экспорт данных в эти форматы либо в файловой системе, либо в сетевой среде.

Но каким образом вся эта информация о программном обеспечении относится к работе сетевого инженера? Прежде всего этот уровень стандартизации уже применяется в сетевых протоколах низкого уровня. Такие протоколы, как BGP, OSPF и TCP/IP, проектировались с учетом необходимости единого языка, с помощью которого все сетевые устройства получили возможность обмениваться данными в глобальной распределенной системе – сети интернет. Почти такая же причина стала отправным пунктом для разработки форматов данных, рассматриваемых в этой главе, – предоставить компьютерным системам возможность понимать друг друга и свободно обмениваться данными.

Каждое установленное, сконфигурированное или обновленное устройство буквально «оживил» разработчик программного обеспечения, который учел все перечисленные выше нюансы. Некоторые поставщики сетевых устройств сознательно предоставили механизмы, позволяющие операторам взаимодействовать с устройством с использованием широко распространенных форматов данных, другие поставщики не уделили этому никакого внимания. Цель данной главы – помочь вам понять важное значение таких стандартизированных и простых форматов данных, научить вас пользоваться их преимуществами для более эффективной автоматизации сети.

Некоторые конфигурационные модели очень хорошо стыкуются с методами автоматизации, например представление модели конфигурации в таких фор-

Tlgm: @it_boooks

матах, данных как XML или JSON. Без каких-либо затруднений можно создать XML-представление определенного набора данных, например в Junos:

```
root@vsrx01> show interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/12.1X47/
    junos-interface" junos:style="normal">
        <physical-interface>
            <name>ge-0/0/0</name>
            <admin-status junos:format="Enabled">up</admin-status>
            <oper-status>up</oper-status>
            <local-index>134</local-index>
            <snmp-index>507</snmp-index>
            <link-level-type>Ethernet</link-level-type>
            <mtu>1514</mtu>
            <source-filtering>disabled</source-filtering>
            <link-mode>Full-duplex</link-mode>
            <speed>1000mbps</speed>

... вывод сокращен в целях экономии места ...
```

Конечно, это представление является не самым простым для восприятия человеком, но суть не в этом. С точки зрения программирования этот формат идеален, так как каждый фрагмент данных размещен в собственном, легко распознаваемом поле. Модулю программы не приходится гадать, где найти имя интерфейса, – оно обозначено общеизвестным и тщательно документированным тегом «name». Здесь очень важно понять различия между тем, как программная система должна взаимодействовать с инфраструктурой компонентов, и восприятием этой инфраструктуры человеком, работающим в командной строке (CLI).

На более высоком уровне восприятия и интерпретации форматов данных в первую очередь важно точно понимать, каким образом мы намерены использовать форматы данных, предоставленных в наше распоряжение. Каждый формат был создан для определенного варианта практического применения, поэтому хорошее понимание этих вариантов применения поможет выбрать вариант, требуемый в данном конкретном случае.

Типы данных

После обсуждения вариантов применения форматов данных важно кратко рассмотреть типы данных, которые могут быть представлены этими форматами. Главная цель форматов – передача таких объектов, как слова, числа и даже более сложные составные объекты между экземплярами программ. Если вы посещали какой-либо курс программирования, то наверняка знакомы с такими типами данных.



Отметим, что начиная с этой главы не рассматриваются какие-либо конкретные языки программирования, все примеры являются обобщенными. Рассматриваемые далее типы данных могут быть представлены различными наименованиями в зависимости от их конкретной реализации.

Tlgm: @it_boooks

В главе 4 рассматривались особенности языка программирования Python, поэтому при необходимости можно вернуться к содержимому этой главы для уточнения определений и вариантов использования типов данных, специфичных для Python.

- *Строка (string)* – вероятно, это самый главный, основополагающий и наиболее часто используемый тип данных. Строка чаще всего применяется для представления последовательности чисел, букв и прочих символов. Если необходимо представить предложение на любом естественном языке в одном из форматов данных, обсуждаемых в этой главе, или в любом языке программирования, то для этого, вероятнее всего, будет использована строка. В языке Python для этого применяется тип данных `str` или `unicode`.
- *Целое число (integer)* – это действительно числовое значение, то есть тип данных для работы с целыми числами, и большинство людей при обсуждении числовых типов данных в первую очередь вспоминают именно целые числа. Это то, что все вы учили в школе: целое число, положительное или отрицательное. Существуют и другие типы числовых данных, такие как числа с плавающей точкой или десятичные дроби, которые можно использовать для описания нецелых значений. В языке Python для представления целых чисел предназначен тип `int`.
- *Логическое значение (boolean)* – один из самых простых типов данных, в котором значениями могут быть либо «истина» (true), либо «ложь» (false). Это широко распространенный тип данных, который в программировании позволяет получить логический результат выполнения некоторой операции, например сравнения двух чисел друг с другом. В языке Python это тип `bool`.
- *Комплексные структуры данных* – данные различных типов могут быть объединены в сложных составных структурах. Все форматы, которые мы будем рассматривать в этой главе, поддерживают основную концепцию, известную как массив (array) или список (list) в некоторых случаях. Это список значений или объектов, к которым можно обращаться по индексу. Кроме того, это может быть список пар ключ-значение, обозначаемый различными наименованиями: словарь, хеш, хеш-отображение, хеш-таблица или просто отображение. Такая структура похожа на массив, но организована в форме пар ключ-значение, в которых ключ и значение могут принадлежать к одному из типов данных, например строка, целое число и т. д. В языке Python массив представлен в нескольких формах: множества, кортежи, списки, но все они представляют последовательность элементов, хотя и отличаются друг от друга разнообразными методами обработки, которые они предлагают. Список пар ключ-значение представлен типом `dict`.

Это далеко не полный список, но в нем перечислено подавляющее большинство типов данных, используемых в практических примерах текущей главы. Еще раз отметим, что подробности, присущие конкретной реализации этих типов данных, действительно зависят от контекста их применения. Также от-

Tlgm: @it_boooks

метим, что все форматы данных, рассматриваемые в текущей главе, широко распространены и имеют мощную поддержку.

В дальнейшем типы данных будут обозначаться моноширинным шрифтом, например `string` или `boolean`, чтобы читателю легче было выделить имя типа данных в обычном тексте.

Мы выяснили, что такое форматы данных, для чего они нужны и какие типы данных могут быть представлены с их помощью, поэтому можно переходить к конкретным примерам и рассматривать реализацию теоретических концепций на практике.

YAML

Если вы читаете эту книгу, потому что уже наблюдали убедительные примеры автоматизации сети в действии или на презентации и хотите узнать об этом больше, то, возможно, вы уже слышали термин YAML. Дело в том, что YAML – это особенно удобочитаемый для человека формат данных, поэтому он рассматривается самым первым в текущей главе.



YAML – это аббревиатура фразы «YAML Ain't Markup Language», сообщающей о том, что создатели формата YAML не хотели ограничиться всего лишь новым стандартом языка разметки, а сделали попытку представить данные в форме, удобной для чтения человеком. Кроме того, аббревиатура YAML является рекурсивной.

Краткий обзор основ YAML

Если сравнить YAML с другими форматами данных, рассматриваемых в текущей главе, например с XML или JSON, то можно заметить, что YAML, как и другие форматы, работает с такими конструкциями, как списки, пары ключ-значение, строки и целые числа. Но вскоре вы увидите, что YAML представляет эти конструкции в наиболее удобной для человека форме. Данные в формате YAML очень легко читать и записывать, если вы хорошо понимаете концепцию типов данных, описанную в предыдущем разделе.

Основная причина увеличения количества инструментальных средств, использующих формат YAML (см. Ansible), – метод определения рабочего потока автоматизации или представления набора данных для работы с ним (подобно списку виртуальных сетей VLAN). Очень удобно использовать формат YAML для создания «с нуля» функционального рабочего потока автоматизации или для определения данных, которые необходимо передать в устройство.

На момент написания этой книги самой свежей версией спецификации являлась YAML 1.2, опубликованная на сайте http://www.yaml.org/. На этом сайте также размещен список программных проектов, реализующих спецификацию YAML, обычно для обеспечения возможности чтения данных в структуры, определяемые конкретным языком программирования, и дальнейшей обработки данных в таких структурах. Если вы предпочитаете пользоваться одним из языков программирования, то полезно будет внимательно изучить приме-

Tlgm: @it_boooks

ры использования YAML в текущей главе и попытаться реализовать их на «своем» языке, возможно, с применением вспомогательных библиотек.

Рассмотрим несколько примеров. Предположим, что необходимо использовать YAML для представления списка поставщиков сетевого оборудования. Если вы внимательно изучили предыдущий раздел, то, вероятнее всего, решите воспользоваться типом данных `string` для представления имен поставщиков и будете совершенно правы. Пример очень прост:

```
---
- Cisco
- Juniper
- Brocade
- VMware
```



Обратите внимание на три дефиса (`---`) в самой первой строке каждого примера в этом разделе – по соглашению YAML это обозначение начала каждого YAML-документа.
В спецификации YAML также определено, что многоточие (...) используется для обозначения конца документа, поэтому можно многократно применять комбинацию из трех дефисов для обозначения нескольких документов в одном файле или в одном потоке данных. Эти методы обычно используются только в каналах связи (например, для обозначения завершения сообщения), то есть это не самый распространенный вариант применения, поэтому в текущей главе мы не будем пользоваться такими способами.

Приведенный в примере YAML-документ содержит четыре элемента. Нам известно, что каждый элемент принадлежит к типу данных `string`. Одним из явных удобств формата YAML является то, что обычно не требуются одиночные или двойные кавычки для обозначения строки, как правило, строки автоматически распознаются YAML-парсером (например, PyYAML). Перед каждым элементом размещен дефис. Поскольку все четыре строки расположены на одном уровне (без смещения), можно считать, что эти строки образуют список длиной в 4 элемента.

По своей гибкости формат YAML очень похож на структуры данных языка Python, поэтому можно пользоваться всеми преимуществами такой гибкости без написания кода на Python. Хороший пример, демонстрирующий универсальность формата, – объединение различных типов данных в одном списке (не все языки программирования поддерживают такое объединение):

```
---
- Core Switch
- 7700
- False
- ['switchport', 'mode', 'access']
```

В этом примере список тоже состоит из 4 элементов. Но типы всех элементов абсолютно различны. Первый элемент `Core Switch` – строка, тип `string`. Второй элемент `7700` интерпретируется как целое число, тип `integer`. Третий – `False` – логическое значение, то есть `boolean`. Эту «интерпретацию» выполняет синтаксический анализатор (парсер) формата YAML, например PyYAML. В частности,

Tlgm: @it_boooks

PyYAML весьма успешно выполняет операцию определения типа данных, которую пользователь применил для обмена информацией.



Логические типы `boolean` формата YAML в действительности очень вариативны и принимают широкий спектр значений, которые в конечном итоге означают одно и то же при интерпретации их парсером YAML.
Например, в приведенном выше примере можно было бы написать не только `False`, но также `no`, `off` или даже просто `n`. Все эти варианты в конечном итоге означают одно и то же: логическое значение `False`. Это главная причина того, что YAML часто используется как пользовательский интерфейс в многих программных проектах.

Четвертый элемент в примере фактически является списком, содержащим три элемента типа string. Это первый пример использования вложенных структур данных в формате YAML. Кроме того, в примере показаны различные способы представления данных. Во «внешнем» списке элементы расположены в отдельных строках и перед каждым элементом размещен дефис. Внутренний список представлен одной строкой с использованием квадратных скобок и запятых. Это два способа записи одного и того же объекта – списка.



Отметим, что иногда имеется возможность помочь парсеру точно определить тип данных, которые предполагается передавать. Например, если бы нужно было интерпретировать второй элемент как строку `string`, а не как целое число `integer`, то его следовало бы записать в двойных кавычках (`"7700"`). Другая причина применения двойных кавычек – случай, когда строка string содержит символ(ы), являющий(е)ся частью синтаксиса YAML, например двоеточие (:).
Получить более подробную информацию обо всех тонкостях синтаксиса можно из официальной документации по конкретной используемой реализации парсера YAML.

Ранее в текущей главе были кратко упомянуты пары ключ-значение (или словари, как они называются в языке Python). YAML без проблем поддерживает эту структуру. Рассмотрим, как можно представить словарь в виде четырех пар ключ-значение:

```
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Здесь ключи показаны как строки `string` в левом столбце, а соответствующие ключам значения размещены справа. Если нужно найти одно из таких значений, например в программе на языке Python, то потребуется ссылка на ключ, соответствующий искомому значению.

Подобно спискам, словари чрезвычайно универсальны в отношении к типам данных хранимых значений. В показанном выше примере сохраняется несколько различных типов данных как значения в парах.

Tlgm: @it_boooks

Также следует отметить, что словари YAML, как и списки, можно записывать несколькими способами. С точки зрения представления данных предыдущий пример абсолютно равнозначен следующему:

```
---
{Juniper: Also a plant, Cisco: 6500, Brocade: True,
VMware: ['esxi', 'vcenter', 'nsx']}
```

Большинство парсеров будет интерпретировать оба этих документа YAML совершенно одинаково, но первый явно более удобен для чтения человеком. При выборе формы представления суть такова: если необходим более удобный для чтения документ, то используйте более подробный, развернутый вариант. В противном случае, возможно, даже не требуется именно формат YAML, можно воспользоваться такими форматами, как JSON или XML. Например, в программных интерфейсах API удобство чтения не играет почти никакой роли, здесь чрезвычайно важна скорость и широкая поддержка ПО.

Для комментариев используется символ хеш («решетка») (#). Комментарий можно размещать как на отдельной строке, так и после данных в той же строке.

```
---
- Cisco    # ocsiC
- Juniper  # repinuJ
- Brocade  # edacorB
- VMware   # erawMV
```

Все символы после # парсер YAML игнорирует.

Как вы сами убедились, YAML вполне можно использовать для обеспечения удобного способа взаимодействия человека с программными системами. Но YAML является относительно новым форматом данных. В области обмена данными непосредственно между программными компонентами (то есть без взаимодействия с человеком) гораздо более распространены другие форматы, такие как XML и JSON, которые к тому же оснащены более тщательно проработанными инструментальными средствами для выполнения поставленных перед ними задач.

Работа с YAML в коде Python

Рассмотрим конкретный пример, наглядно показывающий, как именно интерпретатор YAML считывает и обрабатывает данные, записанные в документе YAML. Воспользуемся структурой данных в формате YAML из предыдущего раздела с элементами различных типов:

```
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Tlgm: @it_boooks

Предположим, что этот документ YAML сохранен в локальной системе в файле с именем *example.yml*. Наша задача: использовать Python для считывания этого YAML-файла, синтаксического разбора его содержимого и представления содержащихся в нем данных в виде переменных соответствующих типов.

Эта задача решается с легкостью, если в дополнение к обычному коду Python воспользоваться ранее упоминаемым парсером YAML от независимого производителя – PyYAML:

```
import yaml
with open("example.yml") as f:
    result = yaml.load(f)
    print(result)
    type(result)
{'Brocade': True, 'Cisco': 6500, 'Juniper': 'Also a plant',
'VMware': ['esxi', 'vcenter', 'nsx']}
<type 'dict'>
```



Во фрагменте кода Python из предыдущего примера используется модуль `yaml`, который устанавливается в комплекте пакета `pyyaml`. Этот пакет легко можно установить с помощью программы `pip`, воспользовавшись указаниями, данными в главе 4.

Приведенный пример показывает, насколько легко можно загрузить содержимое YAML-файла в словарь языка Python. Сначала используется менеджер контекста, чтобы открыть файл для чтения (общий метод для чтения любого текстового файла в языке Python). Затем функция `load()` из модуля `yaml` позволяет загрузить содержимое открытого файла непосредственно в словарь с именем `result`. Следующие две строки выводят содержимое и тип словаря, подтверждая, что все операции были выполнены корректно.

Модели данных в YAML

Во вводной части этой главы было сказано, что модели данных определяют структуру для хранения данных в конкретном формате, таком как YAML, XML или JSON. Рассмотрим один из примеров, приведенных в предыдущих разделах, с точки зрения моделей данных, используемых в YAML.

Предположим, что имеются данные, хранящиеся в формате YAML:

```
---
Juniper: vSRX
Cisco: Nexus
Brocade: VDX
VMware: NSX
```

При взгляде на эти данные YAML человек интуитивно понимает, что видит список производителей сетевого оборудования и соответствующий продукт от каждого производителя. При этом мысленно создается модель данных, в которой каждая запись YAML-документа должна содержать пару значений типа

Tlgm: @it_boooks

string: первая строка (ключ) – имя производителя оборудования, вторая строка (значение) – наименование продукта. В совокупности эти строки формируют словарь, состоящий из пар ключ-значение.

Но если бы мы работали с этой (предполагаемой) моделью данных и получили бы следующие данные:

```
---
Juniper: vSRX
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Это корректная запись документа YAML, но сами данные некорректны. Даже если бы компания Brocade предлагала продукт под названием «True», большинство интерпретаторов YAML прочитает эти данные как логическое значение типа boolean, а не как строку (string). Когда наше программное обеспечение начинает обработку считанных данных, для этого элемента ожидается тип string, но вместо этого проступает значение типа boolean. Вероятнее всего, программа выдаст неверные результаты или даже аварийно завершится.

Модель данных – это способ определения структуры и содержимого данных, хранящихся в конкретном формате, например в YAML. Используя модель данных, можно в явной форме определить, что данные в YAML-документе обязательно должны быть списком пар ключ-значение, а каждое значение непременно должно быть строкой (тип string).

К сожалению, YAML не предоставляет какой-либо встроенный механизм для описания или строгого определения моделей данных. Для этого существуют инструментальные средства от независимых производителей (один из примеров – Kwalify (http://www.kuwata-lab.com/kwalify)). Это одна из причин, по которой формат YAML очень хорошо подходит для взаимодействия человека с машиной, но не всегда пригоден для взаимодействия между машинами (или между программными системами).



YAML считается расширением («надмножеством») формата JSON, который будет рассматриваться в одном из следующих разделов текущей главы. Теоретически это означает, что инструментальные средства для проверки корректности (валидации) схемы JSON, то есть модели данных для документа JSON, также могут применяться для проверки корректности документа YAML.

Формат данных XML предоставляет некоторые функциональные возможности и свойства, которые делают его более подходящим для взаимодействия между машинами (или программными системами). Рассмотрим этот формат более подробно.

Tlgm: @it_boooks

XML

Как было отмечено в предыдущем разделе, если YAML больше подходит для взаимодействия человека с компьютером, то другие форматы, такие как XML и JSON, чаще выбирают для представления данных, когда необходим обмен информацией между компонентами программного обеспечения. В данном разделе рассматривается формат данных XML, а также объясняется, почему он пригоден в этом случае.



Формат XML полностью поддерживается разнообразными инструментальными средствами и языками программирования, например для языка Python имеется библиотека LXML (http://lxml.de/). По существу, само определение XML сопровождается связанными с ним определениями таких сущностей, как схема обязательного выполнения, преобразования и расширенные запросы. В текущем разделе мы всего лишь пытаемся пробудить интерес к XML и рекомендуем самостоятельно поработать с рассматриваемыми здесь инструментальными средствами и форматами данных.

Основы XML

У формата XML и у рассмотренного выше формата YAML имеются некоторые похожие свойства. Например, оба формата по своей сути являются иерархическими. Можно с легкостью вставить группу данных в родительскую структуру:

```
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <osver>NXOS 6.1</osver>
</device>
```

В этом примере элемент `<device>` называется корневым, или просто корнем (root). Несмотря на то что пробелы и отступы (для выравнивания) никак не влияют на корректность XML, с их помощью можно легко выделить самый первый внешний XML-тег в документе. Кроме того, этот тег является родительским (parent) для вложенных в него элементов: `<vendor>`, `<model>` и `<osver>`. Эти элементы называют потомками (children) элемента `<device>`, а по отношению друг к другу они являются «родными братьями» (или сестрами) (siblings). Это очень удобно для хранения метаданных о сетевых устройствах, как можно видеть в приведенном примере. В документе XML может быть несколько экземпляров тега `<device>` (то есть несколько элементов `<device>`), возможно, в свою очередь, вложенных в более обобщенный тег `<devices>`.

Сразу отметим, что каждый элемент-потомок также содержит данные. Корневой элемент содержит элементы-потомки XML, в которых размещены текстовые данные. Вспоминая содержимое раздела о типах данных, можно вполне обоснованно предположить, что эти текстовые данные должны быть представлены значениями типа `string`, например в программе на языке Python.

Кроме того, элементы XML могут иметь атрибуты (attributes):

```
<device type="datacenter-switch" />
```

Tlgm: @it_boooks

Если с каким-либо фрагментом информации связаны некоторые метаданные, то для их размещения не вполне подходит элемент-потомок, вместо этого они размещаются в атрибуте.

В спецификации XML также определена реализация пространства имен, позволяющая избегать конфликтов имен элементов. Разработчики могут использовать любые имена при создании документов XML. При обработке программным компонентом данных в формате XML возможно существование двух элементов с одинаковыми именами, но эти элементы должны иметь различное содержимое и предназначение.

Например, в документе XML может встретиться следующая запись:

```
<device>Palm Pilot</device>
```

Здесь используется имя элемента `<device>`, но становится ясно, что этот элемент не представляет сетевое устройство, а предназначен для другой цели, следовательно, его смысловое значение совершенно отличается от значения определения коммутатора, приведенного в самом начале раздела.

В решении этой проблемы могут помочь пространства имен, позволяющие определять и применять префиксы в документе XML с помощью ключевого слова `xmlns`:

```
<root>
  <e:device xmlns:c="http://example.org/enduserdevices">Palm Pilot</e:device>
  <n:device xmlns:m="http://example.org/networkdevices">
    <n:vendor>Cisco</n:vendor>
    <n:model>Nexus 7700</n:model>
    <n:osver>NXOS 6.1</n:osver>
  </n:device>
</root>
```

Создание и чтение корректного документа XML связано со множеством нюансов. Для ознакомления с ними рекомендуется изучить документацию по XML от W3Schools (http://www.w3schools.com/xml/).

Использование определения схемы XML Schema Definition (XSD) для моделей данных

В формате YAML имеются встроенные конструкции для поддержки описания типов содержащихся в документе данных (дефисы и отступы), но в XML нет подобных механизмов. Многие парсеры XML не принимают во внимание какие-либо предположения или допущения, которые, например, делает PyYAML и другие парсеры YAML.

Напомним, что в начале главы форматы данных были определены как средства, позволяющие программным приложениям и аппаратным (в частности, сетевым) устройствам обмениваться информацией стандартизированными способами. XML представляет собой одно из таких стандартизированных средств обмена информацией. Но форматы данных, подобные XML, не определяют точно, какого рода (kind) данные содержатся в разнообразных полях.

Tlgm: @it_boooks

Чтобы обеспечить правильное размещение данных определенного рода в соответствующих элементах XML, используется определение схемы XML Schema Definition (XSD).

Определение схемы XML Schema Definition (XSD) (http://www.w3schools.com/schema/) позволяет описывать компоновочные блоки, из которых формируется документ XML. Используя язык определения схемы, можно сформулировать ограничения, описывающие, где именно должны (или не должны) размещаться те или иные данные в конкретном документе XML. Попытки обеспечить такую функциональность предпринимались и раньше (например, DTD), но возможности этих механизмов были ограниченными. Кроме того, XSD написан собственно на языке XML, что существенно упрощает использование этого механизма.

Одним из наиболее часто применяемых на практике вариантов использования XSD (а также схемы любого типа или любого языка моделирования) является генерация исходного кода структур данных, соответствующих определенной схеме. Затем этим исходным кодом можно воспользоваться для автоматической генерации документа XML, совместимого с той же схемой, вместо формирования документа вручную.

Чтобы продемонстрировать, как это можно сделать на языке Python, еще раз обратимся к примеру XML-описания коммутатора.

```
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <osver>NXOS 6.1</osver>
</device>
```

Наша задача – вывести этот документ XML в консоли. Это можно сделать следующим образом: сначала создать документ XSD, затем сгенерировать из этого документа код на языке Python с помощью инструмента от независимого производителя. После этого программный код используется для вывода в консоли требуемого документа XML.

Сначала создадим файл схемы XSD, в котором содержится описание данных, которые необходимо вывести в консоли:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:element name="device">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vendor" type="xs:string"/>
      <xs:element name="model" type="xs:string"/>
      <xs:element name="osver" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Tlgm: @it_boooks

В этом документе XSD определено, что каждый элемент <device> может содержать три элемента-потомка, а данные в каждом из потомков обязательно должны быть строками (тип string). Здесь не показана поддерживаемая спецификацией XSD возможность определения обязательного присутствия конкретного элемента-потомка. Другими словами, можно определить, что элемент <device> обязательно должен содержать элемент-потомок <vendor>.

Можно воспользоваться инструментальным средством pyxb для создания файла с исходным кодом Python, содержащим объект класса, представляющего описанную выше схему:

```
~$ pyxbgen -u schema.xsd -m schema
```

В текущем каталоге создается файл schema.py. Если сейчас войти в интерактивную оболочку Python, то можно импортировать этот файл схемы и работать с его содержимым. В следующем примере создается экземпляр сгенерированного объекта, устанавливаются некоторые его свойства, затем выполняется преобразование в формат XML с помощью функции toxml():

```
import schema
dev = schema.device()
dev.vendor = "Cisco"
dev.model = "Nexus"
dev.osver = "6.1"
dev.toxml("utf-8")
'<?xml version="1.0" encoding="utf-8"?><device><vendor>Cisco</vendor><model>Nexus
</model><osver>6.1</osver></device>'
```

Это лишь один из способов решения поставленной задачи, поскольку существуют независимые библиотеки, позволяющие генерировать код для XSD-файлов. Также следует обратить внимание на инструмент generateDS, размещенный на сайте https:// pypi.python.org/pypi/generateDS/.



Некоторые прикладные программные интерфейсы API RESTful (см. главу 7) используют XML для кодирования данных, передаваемых между внешними компонентами программ. Применение XSD позволяет разработчику генерировать совместимый код XML намного более точно и с меньшим количеством вспомогательных операций. Если в одном из сетевых устройств вам придется работать с RESTful API, то запросите у производителя документацию на схему XSD, и вы сэкономите немало времени.

Более подробное и полное описание XSD можно найти на сайте W3School: https://www.w3.org/standards/xml/schema.

Преобразование XML с помощью XSLT

В большинстве физических сетевых устройств пока еще преобладает основанный на тексте и ориентированный на взаимодействие с человеком механизм конфигурации, поэтому, вероятно, вы знакомы с некоторыми типами формата шаблонов. Существует огромное количество таких форматов, а шаблоны (templates) вообще очень удобны для реализации безопасной и эффективной автоматизации сети.

Tlgm: @it_boooks

В следующей главе (глава 6) будут более подробно рассматриваться языки описания шаблонов, а особое внимание будет уделено Jinja. Но сейчас, когда мы обсуждаем XML, все же необходимо дать краткое описание языка Extensible Stylesheet Language Transformations (XSLT).

XSLT – это язык для выполнения преобразований в формат данных XML, главным образом в специализированный формат XHTML или в другие документы XML. Как и для многих других языков, связанных с XML, описание XSLT можно найти на сайте W3School (http://www.w3schools.com/xsl/).

Рассмотрим практический пример заполнения шаблона XSLT осмысленными данными, чтобы в результате получился содержательный документ. Как и в предыдущих примерах, для этого будет использоваться код Python.

Сначала нужны исходные текстовые данные для заполнения шаблона. Например, следующий документ XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

Здесь формируется список авторов (тег `<authors>`), при этом для каждого автора (тег `<author>`) в отдельных элементах записывается имя `<firstName>` и фамилия `<lastName>`. Наша задача – использовать эти данные для генерации таблицы, содержащей имена и фамилии авторов, на языке разметки HTML с помощью документа XSLT.

Шаблон XSLT для выполнения этой задачи может выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output indent="yes"/>
<xsl:template match="/">
  <html>
  <body>
  <h2>Authors</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th style="text-align:left">First Name</th>
        <th style="text-align:left">Last Name</th>
```

Tlgm: @it_boooks


```
      </tr>
      <xsl:for-each select="authors/author">
      <tr>
        <td><xsl:value-of select="firstName"/></td>
        <td><xsl:value-of select="lastName"/></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

По приведенному выше документу XSLT необходимо сделать несколько замечаний:

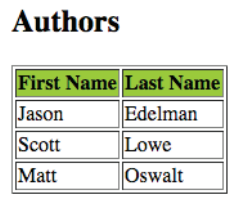
- отметим наличие конструкции простого цикла `for-each` в коде, похожем на обычный корректный код HTML. Это прием, весьма распространенный в языке шаблонов, – статический текст остается статическим, а в необходимых местах размещаются небольшие логические выражения. Более подробно об этом вы узнаете в главе 6;
- следует также отметить, что оператор цикла `for-each` использует аргумент-«координату» (указанный как `"authors/author"`) для точного указания той части документа XML, которую необходимо использовать. Такой аргумент называется XPath, а его синтаксис применяется в документах XML и в инструментальных средствах для определения точной позиции в дереве XML;
- кроме того, используется оператор `value-of` для динамической подстановки (подобной подстановке значения переменной в программах на языке Python) текстового значения, взятого из данных XML.

Предположим, что приведенный выше шаблон XSLT сохранен в файле *template.xsl*, а данные об авторах – в файле *xmldata.xml*. Теперь можно вернуться в интерактивный интерпретатор Python для объединения двух созданных компонентов и формирования итогового вывода в формате HTML.

```
from lxml import etree
xslRoot = etree.fromstring(open("template.xsl").read())
transform = etree.XSLT(xslRoot)
xmlRoot = etree.fromstring(open("xmldata.xml").read())
transRoot = transform(xmlRoot)
print(etree.tostring(transRoot))
<html><body><h2>Authors</h2><table border="1"><tr bgcolor="#9acd32">
<th style="text-align:left">First Name</th><th style="text-align:left">Last Name
</th></tr>
<tr><td>Jason</td><td>Edelman</td></tr>
<tr><td>Scott</td><td>Lowe</td></tr>
<tr><td>Matt</td><td>Oswalt</td></tr></table></body></html>
```

В результате выполнения этого кода генерируется требуемая HTML-таблица, показанная на рис. 5.1.

Tlgm: @it_boooks



Authors

First Name	Last Name
Jason	Edelman
Scott	Lowe
Matt	Oswalt

Рис. 5.1 ❖ Таблица в формате HTML, сгенерированная с помощью шаблона XSLT

В шаблонах XSLT также можно применять следующие логические операторы:

- `<if>` – позволяет выводить заданный(е) элемент(ы) только в случае выполнения указанного условия;
- `<sort>` – сортирует элементы перед выводом;
- `<choose>` – расширенная версия оператора if (обеспечивает выполнение ветвей else if или else).

Можно продолжить работу с этим примером и воспользоваться основной концепцией для создания шаблона сетевой конфигурации, определив конфигурационные данные в документе XML и соответствующий шаблон XSLT, как показано в примерах 5.1 и 5.2.

Пример 5.1 ❖ Данные об интерфейсах в формате XML

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaces>
    <interface>
        <name>GigabitEthernet0/0</name>
        <ipv4addr>192.168.0.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/1</name>
        <ipv4addr>172.16.31.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/2</name>
        <ipv4addr>10.3.2.1 255.255.254.0</ipv4addr>
    </interface>
</interfaces>
```

Пример 5.2 ❖ Шаблон XSLT для создания конфигурации маршрутизатора

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.example.org/routerconfig">
<xsl:template match="/">
    <xsl:for-each select="interfaces/interface">
      interface <xsl:value-of select="name"/><br />
          ip address <xsl:value-of select="ipv4addr"/><br />
    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Tlgm: @it_boooks

Используя документы XML и XSLT, показанные в примерах 5.1 и 5.2, можно получить простейшую конфигурацию маршрутизатора тем же способом, который применялся для генерации HTML-страницы:

```
interface GigabitEthernet0/0
ip address 192.168.0.1 255.255.255.0
interface GigabitEthernet0/1
ip address 172.16.31.1 255.255.255.0
interface GigabitEthernet0/2
ip address 10.3.2.1 255.255.254.0
```

Здесь наглядно продемонстрирована возможность создания сетевой конфигурации с использованием XSLT. Но, по общему признанию, этот способ не совсем оптимален. Вероятнее всего, вы обнаружите, что Jinja является более удобным языком шаблонов для создания сетевых конфигураций, к тому же этот язык обладает множеством функциональных возможностей, обеспечивающих поддержку автоматизации сети. Язык шаблонов Jinja рассматривается в главе 6.

Поиск в данных XML с использованием XQuery

В предыдущем разделе отмечалось использование XPath в документах XSLT для точного определения места расположения заданных элементов в документе XML. Но если необходимо выполнить более сложную операцию поиска, то потребуется нечто большее, нежели простая система координат.

XQuery применяет инструменты, подобные XPath, для поиска и извлечения данных из документа XML. Например, если обеспечивается доступ через REST API маршрутизатора или коммутатора с использованием языка Python, то можно написать дополнительный фрагмент кода для получения точно определенной части вывода XML-данных, которую необходимо обработать. Кроме того, можно напрямую использовать XQuery в процессе приема данных, чтобы проверить наличие тех фрагментов данных, которые нужно обработать в Python-программе.

XQuery – это мощный инструмент, который по своей сущности очень похож на язык программирования. Более подробную информацию по XQuery можно найти на сайте W3School (https://www.w3.org/standarts/xml/query.html).

JSON

Итак, мы рассмотрели YAML – инструмент, наиболее подходящий для взаимодействия человека с компьютером, обеспечивающий импорт в структуры данных многочисленных языков программирования. Мы также рассмотрели XML, не самый удобный формат для восприятия человеком, но обладающий богатой инфраструктурой с многочисленными инструментальными средствами и мощной поддержкой программного обеспечения. В этом разделе рассматривается JSON, объединяющий некоторые преимущества вышеупомянутых инструментов в едином формате данных.

Tlgm: @it_boooks

Основы формата JSON

Формат JSON появился в то время, когда веб-разработчикам требовался простой и удобный механизм обмена информацией между веб-серверами и аплетами, встроенными в веб-страницы. В это же время появился и формат XML, но его чрезмерная громоздкость не соответствовала требованиям «моментально доступного» интернета.

Кроме того, следует также отметить, что форматы YAML и XML существенно отличаются друг от друга в том, как они выполняют преобразование данных в модели данных большинства языков программирования, подобных Python. Импортирование документа YAML в исходный код с использованием библиотеки PyYAML или других инструментальных бибилиотек выполняется практически без лишних трудозатрат. Но обработка данных в формате XML обычно требует выполнения нескольких дополнительных операций в зависимости от того, что вы хотите сделать.

Именно поэтому и по нескольким другим причинам в начале 2000-х годов появился формат JavaScript Object Notation (JSON) и сразу получил широкое распространение. Он был задуман как упрощенная версия XML, более подходящая для моделей данных, принятых в большинстве языков программирования. Кроме того, JSON считался более удобным для чтения человеком, хотя это не самая главная характеристика форматов данных.



Отметим, что большинство считает JSON подмножеством YAML. В действительности многие широко используемые парсеры YAML также могут выполнять синтаксический анализ данных в формате JSON, как если бы это были данные YAML. Тем не менее некоторые подробности этой взаимосвязи сложны и требуют внимательного изучения. Более подробную информацию по этому вопросу можно найти в соответствующем разделе спецификации YAML (http://yaml.org/spec/1.2/spec.html#id2759572).

В предыдущем разделе был показан пример представления имен трех авторов в документе XML:

```
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

Tlgm: @it_boooks

Чтобы наглядно продемонстрировать различие между JSON и XML, особенно выделив при этом более простую сущность JSON, ниже приведена аналогичная модель данных в формате JSON:

```
{
    "authors":[
        {
            "firstName": "Jason",
            "lastName": "Edelman"
        },
        {
            "firstName": "Scott",
            "lastName": "Lowe"
        },
        {
            "firstName": "Matt",
            "lastName": "Oswalt"
        }
    ]
}
```

Это выглядит значительно проще, чем представление тех же данных в формате XML. Неудивительно, что JSON стал более привлекательным в начале 2000-х годов, когда начиналась эпоха «Web 2.0».

Рассмотрим более подробно некоторые особенности формата, использованного в последнем примере. Сразу отметим, что весь блок данных заключен в фигурные скобки {}. Это общее правило, сообщающее о том, что объекты JSON содержатся внутри скобок. Можно считать «объекты» парами ключ-значение или словарями, подобными словарям, которые мы рассматривали в разделе YAML. Объекты JSON всегда используют значения типа `string` для описания ключей в своих конструкциях.

В приведенном примере ключом является строка `"authors"`, а значением для этого ключа служит список JSON, который также практически равнозначен списку в формате YAML, то есть представляет собой упорядоченную последовательность из нуля и более значений. Список обозначен квадратными скобками [].

В списке содержатся три объекта (разделенных запятыми и символами перехода на новую строку), каждый из которых состоит из двух пар ключ-значение. Первая пара описывает имя автора (ключ `"firstName"`), вторая пара описывает фамилию автора (ключ `"lastName"`).

В начале главы были приведены характеристики основных типов данных, но сейчас мы кратко рассмотрим типы данных, поддерживаемых в формате JSON. Они в определенной степени похожи на типы данных, применяемых в формате YAML:

- number – десятичное число со знаком;
- string – последовательность символов, например слово или предложение;

Tlgm: @it_boooks

- boolean – логическое значение `True` или `False`;
- array – массив, упорядоченный список значений, в котором могут содержаться элементы различных типов (обозначается квадратными скобками `[]`);
- object – неупорядоченный набор пар ключ-значение, в котором ключи обязательно принадлежат к типу `string` (обозначается фигурными скобками {});
- null – пустое значение, обозначаемое ключевым словом `null`.

Рассмотрим работу с форматом JSON средствами языка Python и способы обработки данных в этом формате. В некоторой степени это напоминает использование словарей Python, описанное в главе 4.

Обработка формата JSON в коде Python

Формат JSON поддерживается многими языками программирования. На практике вам часто будет предоставлена возможность простого импортирования структур данных JSON в конструкции используемого языка программирования однострочной командой. Рассмотрим несколько примеров.

Предположим, что данные в формате JSON хранятся в обычном текстовом файле:

```
{
  "hostname": "CORESW01",
  "vendor": "Cisco",
  "isAlive": true,
  "uptime": 123456,
  "users": {
    "admin": 15,
    "storage": 10,
  },
  "vlans": [
    {
      "vlan_name": "VLAN30",
      "vlan_id": 30
    },
    {
      "vlan_name": "VLAN20",
      "vlan_id": 20
    }
  ]
}
```

Наша задача – импортировать данные из этого файла в конструкции применяемого в текущий момент языка программирования.

Начнем с языка Python, который располагает инструментами для работы с форматом JSON, включенными непосредственно в стандартную библиотеку, а именно в пакет `json`. В приводимом ниже примере определяется структура данных JSON (позаимствованная из статьи Википедии о JSON) внутри программы на языке Python, но эти данные можно легко извлечь из файла или

Tlgm: @it_boooks

с помощью REST API. Как можно видеть, импортирование данных JSON выполняется вполне понятным образом (см. комментарии в коде):

```
# В языке Python имеются очень полезные инструменты для работы с форматом JSON,
# которые являются частью стандартной библиотеки, то есть фактически встроены в язык.
import json

# Можно загрузить содержимое JSON-файла в переменную data
with open("json-example.json") as f:
    data = f.read()

# json_dict - это словарь, а json.loads отвечает за размещение данных JSON
# в этом словаре.
json_dict = json.loads(data)

# Вывод информации о сформированной в результате структуре данных Python
print("The JSON document is loaded as type {0}\n".format(type(json_dict)))
#  "Документ JSON загружается как тип ... "
print("Now printing each item in this document and the type it contains")
#  "Теперь выводится каждый элемент этого документа и тип данных, содержащихся в нем"
for k, v in json_dict.items():
    print(
        "-- The key {0} contains a {1} value.".format(str(k), str(type(v)))
  # "-- Ключ {0} содержит значение {1}."
    )
```

В нескольких последних строках кода можно видеть, как в языке Python интерпретируются и выводятся данные после импортирования. После запуска этой Python-программы получаем следующий результат:

```
~ $ python json-example.py

The JSON document is loaded as type <type 'dict'>

Now printing each item in this document and the type it contains
-- The key uptime contains a <type 'int'> value.
-- The key isAlive contains a <type 'bool'> value.
-- The key users contains a <type 'dict'> value.
-- The key hostname contains a <type 'unicode'> value.
-- The key vendor contains a <type 'unicode'> value.
-- The key vlans contains a <type 'list'> value.
```



Возможно, вы впервые работаете с данными в кодировке Unicode. Для упрощения лучше считать, что такие данные приблизительно соответствуют типу данных `str` (строка), описанному в главе 4.

В языке Python тип `str` в действительности представляет собой последовательность байтов, в то время как стандарт Unicode определяет другую, широко используемую в настоящее время кодировку.

Использование механизма JSON Schema для моделей данных

В разделе о формате YAML была описана концепция моделей данных и отмечено, что в YAML нет встроенных механизмов для формирования моделей данных. В разделе о формате XML рассматривался механизм XSD, позволяющий

Tlgm: @it_boooks

создавать схему (или модель данных) в коде XML, то есть точно определять типы данных, содержащихся в документе XML. А есть ли поддержка моделей данных в JSON?

В формате JSON также имеется механизм формирования схемы – JSON Schema. Его спецификация определена в документе http://json-schema.org/documentation.html, который также был принят как предварительный (draft) стандарт интернета.

Существует реализация механизма JSON Schema на языке Python (https://pypi.python.org/pypi/jsonschema), а также реализации на других языках программирования.

Прежде чем завершить эту главу, рассмотрим один из способов описания моделей данных, независимый от какого-либо конкретного формата данных. XSD работает только для XML, JSON Schema работает только для JSON. Существует ли способ описания модели данных, который можно использовать и для XML, и для JSON? Оказывается, существует, и в следующем разделе мы рассмотрим инструмент для решения этой проблемы: YANG.

Создание моделей данных с использованием YANG

Ранее в текущей главе обсуждались модели данных в контексте конкретных форматов данных, например формирование модели данных в форматах XML и JSON. В этом разделе мы рассмотрим модели данных с более обобщенной точки зрения, затем обсудим применение языка YANG для описания моделей данных, предназначенных специально для сетевой среды.

Для лучшего понимания общих концепций приведем несколько важных фактов, относящихся к моделям данных:

- модели данных описывают ограниченный набор данных в форме языка описания схемы (например, XSD);
- модели данных используют четко определенные типы и параметры, чтобы сформировать структурированное и стандартное представление данных;
- модели данных не выполняют передачу данных и не принимают во внимание используемые транспортные протоколы более низкого уровня (например, как JSON, так и XML можно использовать поверх протокола HTTP).

После вводной информации о моделях данных можно перейти к рассмотрению языка YANG.

Общий обзор языка YANG

YANG – это язык моделирования данных, определенный в документе RFC 6020. В некотором роде он аналогичен языку механизма XSD для данных в формате XML, но YANG специализирован для сетевых конструкций. YANG используется

Tlgm: @it_boooks

для моделирования данных конфигурации и рабочего состояния, а также для моделирования общих данных RPC. Обобщенные данные и операции RPC позволяют моделировать обобщенные задачи, такие как обновление устройства.

Язык YANG предоставляет возможность определения синтаксиса и семантики для упрощения определения данных с использованием встроенных и создаваемых пользователем типов. Можно определить семантику, в соответствии с которой идентификатор VLAN ID обязательно должен находиться в диапазоне от 1 до 4094. Можно определить допустимые рабочие состояния интерфейса, ограниченные исключительно характеристиками up или down. Модель определяет эти типы конструкций и в конечном итоге становится источником правильных решений и допустимых операций в сетевом устройстве.

Существуют различные типы моделей YANG. Некоторые из этих моделей созданы конечными пользователями, другие сформировали производители оборудования или открытые рабочие группы:

- модели, соответствующие отраслевому стандарту, разработанные группами, такими как IETF и OpenConfig Working Group. Эти модели независимы от производителей оборудования и платформы. Каждая модель, созданная группой разработки открытых стандартов, предоставляет базовый набор характеристик для конкретной функции или свойства;
- существуют модели, специально ориентированные на определенного производителя сетевого оборудования. Как известно, почти каждый производитель оборудования предлагает собственное решение для организации объединенного универсального канала связи на основе различных моделей и типов устройств (VSS, VPC, MCLAG, Virtual Chassis). Таким образом, каждому производителю оборудования необходима собственная модель данных, если он намерен соответствовать требованиям архитектуры, управляемой моделью;
- у каждого производителя сетевого оборудования можно обнаружить различия в определенных функциональных характеристиках. Следовательно, существуют модели, ориентированные на конкретные платформы. Может случиться так, что OSPF работает на платформе X совершенно по-другому, нежели на платформе Y того же производителя. Поэтому потребуется другая модель.

Практическое применение языка YANG

Многое можно описать словами. Иногда графическая иллюстрация стоит тысячи слов, поэтому наглядная демонстрация того, как код YANG отображается в формат XML, JSON или в команды для CLI, также должна заменить тысячу слов. Мы начинаем подробно изучать команды языка YANG, чтобы понять, как код YANG преобразуется в данные, которыми обмениваются две системы.



Здесь мы не показываем, как создается специализированная модель на языке YANG, поскольку это тема отдельной книги. Здесь рассматривается лишь несколько команд YANG, что помочь читателям лучше понять работу с языком YANG.

Tlgm: @it_boooks

В языке YANG имеется команда `leaf`, которая позволяет определить объект как единственный экземпляр с одним набором значений, у которого нет объектов-потомков.

```
leaf hostname {
    type string;
    mandatory true;
    config true;
    description "Hostname for the network device";
}
```

Из этого примера, демонстрирующего применение команды `leaf`, можно понять, что делает этот код. Здесь определяется конструкция, предназначенная для хранения значения имени хоста (hostname), размещенного на сетевом устройстве. Этот объект называется `hostname`, он обязательно должен быть строкой, но можно изменять его конфигурацию.

С помощью команд `leaf` можно также определять эксплуатационные рабочие данные, установив для элемента `config` значение `false`.

Команда языка YANG `leaf` в форматах XML и JSON интерпретируется как один элемент или пара ключ-значение.

```
<hostname>NYC-R1</hostname>
{
    "hostname": "NYC-R1"
}
```

Следующая рассматриваемая здесь команда языка YANG – `leaf-list`. Она похожа на команду `leaf`, но с ее помощью можно создавать несколько экземпляров. Поскольку создается объект типа список, в команде имеется параметр `ordered-by`, для которого устанавливается значение `user` или `system`, определяющее критерий упорядоченности – списки ACL, или строки SNMP, или имена серверов.

```
leaf-list name-server {
    type string;
    ordered-by user;
    description "List of DNS servers to query";
}
```

Команда `leaf-list` в форматах XML и JSON представляется в виде одного элемента, как показано ниже (первая запись – формат XML, вторая – формат JSON):

```
<name-server>8.8.8.8</name-server>
<name-server>4.4.4.4</name-server>
```

```
{
    "name-server": [
        "8.8.8.8",
        "4.4.4.4"
    ]
}
```

Tlgm: @it_boooks

Еще одной командой языка YANG является `list`. Эта команда позволяет создавать список из объектов `leaf` и `leaf-list`. Ниже приведен пример определения списка командой `list`.

```
list vlan {
   key "id";
   leaf id {
       type int;
       range 1..4094;
   }
   leaf name {
       type string;
   }
}
```

Здесь очень важно понять, как код на языке YANG моделируется в формате XML и JSON. Ниже приведены примеры представления модели YANG в форматах XML и JSON соответственно.

```
<vlan>
    <id>100</id>
    <name>web_vlan></name>
</vlan>
<vlan>
    <id>200</id>
    <name>app_vlan></name>
</vlan>
```

```
{
    "vlan": [
        {
            "id": "100",
            "name": "web_vlan"
        },
        {
            "id": "200",
            "name": "app_vlan"
        }
    ]
}
```

Это простейший пример для начинающих, который помогает понять, как данные моделируются в языке YANG и как они кодируются в форматах XML и JSON для передачи в сетевой среде.

Последняя рассматриваемая в этом разделе команда языка YANG – `container`. Контейнер напрямую отображается в иерархическую структуру в форматах XML и JSON. В предыдущем примере был показан список VLAN, но в нем не было внешней конструкции или тега XML, который содержал бы все элементы VLAN. Добавим внешний контейнер с именем `vlan`.

```
container vlans {
    list vlan {
```

Tlgm: @it_boooks


```
        key "id";
        leaf id {
            type int;
            range 1..4094;
        }
        leaf name {
            type string;
        }
    }
}
```

Единственное отличие последнего примера от предыдущего – добавление в начало команды `container`. Ниже показаны представления этого объекта-контейнера в форматах XML и JSON.

```
<vlans>
  <vlan>
      <id>100</id>
      <name>web_vlan></name>
  </vlan>
  <vlan>
      <id>200</id>
      <name>app_vlan></name>
  </vlan>
</vlans>
```

```
{
    "vlans": {
        "vlan": [
            {
                "id": "100",
                "name": "web_vlan"
            },
            {
                "id": "200",
                "name": "app_vlan"
            }
        ]
    }
}
```

Цель текущего раздела – не только привести краткие описания нескольких команд YANG, но также наглядно показать, что YANG в действительности представляет собой просто язык моделирования данных. Это один из способов установления ограничений при вводе данных и использования в API введенных данных, уже представленных в формате XML, JSON или в любом другом формате. Далее само устройство проверяет полученные данные на соответствие предварительно определенной модели.

Язык моделирования, подобный YANG, позволяет определять семантику и ограничения для заданного набора данных. Как сетевое устройство может обеспечить нумерацию VLAN в диапазоне от 1 до 4094? Как сетевое устрой-

Tlgm: @it_boooks

ство может установить ограничение на административное состояние: только shutdown или no shutdown? Ответ на подобные вопросы – наличие средств для корректного определения данных. Такие средства определяют схему документа или специализированный язык моделирования. Одним из средств решения этой задачи является механизм XML Schema Definitions, который рассматривался в текущей главе. Несмотря на то что XSD представляет собой эффективный способ определения схемы для XML-документов, его нельзя считать универсальным решением для сетевой среды, поэтому сейчас в сетевой отрасли наблюдается повышение внимания к методикам формирования схем и моделей. YANG – язык моделирования, специально предназначенный для сетевой среды, он хорошо работает с сетевыми конструкциями. Например, в языке YANG имеются встроенные типы для проверки корректности данных, вводимых как IPv4-адреса, BGP AS или MAC-адреса. Кроме того, язык YANG не зависит от какого-либо типа кодирования. Модель может быть написана на YANG, затем представлена в формате JSON или XML. Именно такую методику предлагает RESTCONF. RESTCONF – это REST API, который использует данные в формате XML или JSON, модель для которых определена с помощью языка YANG. RESTCONF подробно рассматривается в главе 7.

Резюме

В этой главе рассматривались некоторые основополагающие концепции работы с данными. Форматы данных, такие как YAML, XML, JSON, используются для кодирования данных при передаче между приложениями, системами и устройствами. Форматы данных определяют, как кодируются данные, но не всегда определяют структуру данных. Модели данных определяют структуру данных в некотором формате. Иногда модели данных связаны с конкретным форматом, например XSD специализирован только для XML, JSON Schema – только для JSON. Язык YANG предоставляет независимый от формата способ описания модели данных, которая затем может быть представлена в форматах XML и JSON.

В следующей главе будут подробно рассматриваться шаблоны и их применение для автоматизации сети.

Tlgm: @it_boooks

Глава 6

Шаблоны сетевой конфигурации

Большая часть работы сетевых инженеров выполняется в командной строке (CLI), при этом в основном применяются специализированные ключевые слова и выражения, которые могут многократно повторяться с небольшими изменениями отдельных параметров. Такая методика работы не только неэффективна, но также существенно увеличивает вероятность совершения ошибок. Например, может быть вполне понятно, как конфигурировать связь с соседними узлами по протоколу граничного шлюза BGP на устройстве Cisco IOS, но далеко не всегда понятны более мелкие, «неинтуитивные» конфигурации, когда необходимо, скажем, помнить о добавлении правильной конфигурации BGP-группы. В сетевой среде чаще всего одну задачу можно решить несколькими различными способами, но каждая организация обычно придерживается одного выбранного способа.

Одним из главных преимуществ автоматизации сети является логическая согласованность, то есть предсказуемость и воспроизводимость одинаковых результатов при внесении изменений в процессе формирования сетевой инфраструктуры, а также гарантированное достижение поставленной цели. Одним из лучших способов обеспечения логической согласованности является создание шаблонов для полностью автоматизированного взаимодействия с сетевой средой.

Создание шаблонов для конкретных сетевых конфигураций означает унификацию этих конфигураций в соответствии со стандартами организации, а также предоставление возможности администраторам сетей и даже пользователям (служба техподдержки, сетевой операционный центр (NOC), IT-инженеры) динамически изменять некоторые параметры сети при необходимости. Такой подход позволяет быстро вносить изменения, при этом требуется гораздо меньше вспомогательной информации, но сохраняется логическая согласованность, целостность, поскольку шаблоны содержат все необходимые конфигурационные команды, определяемые общей стратегией управления сетью.

Tlgm: @it_boooks

В начале главы дается общий обзор инструментов для работы с шаблонами, затем описываются некоторые специализированные реализации шаблонов и рассматривается практическое применение инструментальных средств для создания шаблонов сетевой конфигурации.

Современные языки шаблонов

В действительности технологии использования шаблонов применяются с довольно-таки давних времен. Даже простой поиск по выражению «template languages» выдает огромное количество результатов, в том числе по нескольку вариантов практически для каждого языка программирования, так или иначе связанного с сетевой средой.

Также можно заметить, что подавляющее большинство этих языков активно используется для создания приложений в области веб-разработки. Причина в том, что сама веб-среда основана на шаблонах. Вместо написания многочисленных HTML-файлов для страниц профилей каждого отдельного пользователя соцсети разработчики создают один файл и подставляют динамические значения в этот шаблон в зависимости от данных, предоставляемых внутренними компонентами.

Таким образом, для языков шаблонов имеется обширный набор вариантов практического применения. Самый очевидный вариант – использование в веб-разработке. Разумеется, языки шаблонов применяются для создания сетевой конфигурации, о чем мы будем говорить в текущей главе, но следует отметить, что языки шаблонов применимы в любых приложениях, связанных с обработкой текста, в том числе для создания документации и отчетов.

Важно помнить, что работа с шаблонами разделяется на три составные части, или этапа. Первый этап – создание шаблона. На втором этапе необходима некоторая форма данных, которые в конечном итоге вносятся в шаблон, заполняя его, чтобы получить осмысленный результат, например конфигурацию сети. Логическим продолжением является третий этап – управление данными в шаблоне. Это может быть какой-либо инструмент автоматизации, например Ansible, который будет подробно рассматриваться в главе 9, или можно воспользоваться для управления данными средствами какого-либо языка программирования, например Python, как будет показано в текущей главе. Сами по себе шаблоны, без средств управления данными, не очень полезны.



Большинство языков шаблонов не является полноценными «языками программирования» в общепринятом смысле, чаще всего язык шаблонов тесно связан с другим языком, который будет управлять данными в создаваемых шаблонах. В результате наблюдаются некоторые черты сходства между каждым конкретным языком шаблонов и его «родительским», управляющим языком. Удачный пример такой взаимосвязи мы будем подробно рассматривать в текущей главе: язык шаблонов Jinja появился в среде сообщества пользователей языка программирования Python, поэтому между ними обнаруживается определенное сходство.

Tlgm: @it_boooks

> Если вы выбираете язык шаблонов для практического использования, то, вероятнее всего, самым главным критерием выбора будет «настоящий» язык программирования, с которым вы знакомы лучше всего (не важно, пишете ли вы собственный исходный код или пользуетесь существующими инструментальными средствами, такими как Ansible), под который следует подбирать наиболее подходящий язык шаблонов.

Как уже было отмечено в текущей главе, языки шаблонов не всегда реализуют какую-либо новую концепцию, тем не менее мы постоянно видим, как новые идеи и даже целые языки формируют собственную экосистему. Если обратиться к истории языков шаблонов, то можно обнаружить, что большое количество таких языков было создано для решения самой главной задачи в веб-среде: для создания динамического контента. Эта задача легко решается в наши дни, но во времена, когда веб-среда еще только формировалась, а большинство веб-сайтов создавалось на основе статического содержимого, динамическая загрузка фрагментов данных на страницу стала крупным достижением в веб-разработке.

Использование шаблонов для веб-разработки

Django (https://www.djangoproject.com/) – это мощная программная веб-среда на основе языка Python, которая представляет собой самый яркий современный пример реализации описанной выше концепции. В Django имеется язык шаблонов, позволяющий разработчику создавать веб-контент в соответствии с этой концепцией, но, кроме того, предлагается способ, которым можно сделать динамическими части веб-страницы. Используя язык шаблонов Django, разработчик может создать в статической в целом странице части, в которые будут динамически загружаться данные, когда пользователь обратится к этой странице.

Ниже приведен очень простой пример, который выглядит почти как HTML-документ, но содержит несколько элементов, заменяемых с помощью подстановки значений переменных (эти элементы обозначены двойными фигурными скобками {{ }}).

```
<h1>{{ title }}</h1>
{% for article in article_list %}
<h2>
  <a href="{{ article.get_absolute_url }}">
    {{ article.headline|upper }}
  </a>
</h2>
{% endfor %}
```

Этот шаблон может быть обработан Django, когда пользователь загружает страницу. При обработке программная среда Django подставляет значения переменных `title` и `article_list`, и пользователь получает страницу, заполненную реальными данными. Разработчику не нужно писать код статической HTML-страницы для каждого варианта, который может потребоваться пользо-

Tlgm: @it_boooks

вателю, содержимое управляется логикой внутреннего компонента этого веб-приложения.



Язык шаблонов Django очень похож (но не идентичен) на язык шаблонов Jinja (http://jinja.pocoo.org/docs/dev/), который обсуждается в текущей главе. Если синтаксис не вполне понятен, не волнуйтесь, немного позже мы подробно рассмотрим все его детали. Но сейчас главное внимание сосредоточено на общих концепциях и преимуществах, которые обеспечивают шаблоны: целостность и согласованность.

Существует множество других языков шаблонов, которые не рассматриваются в текущей главе, тем не менее нужно иметь представление о них. Для того же языка Python имеется несколько альтернативных вариантов, то есть не только ранее упомянутые Django и Jinja, но также Mako (http://www.makotemplates.org/) и Genshi (http://genshi.edgewall.org/). Некоторые языки, например Go (https://golang.org/pkg/text/template/) и Ruby (http://ruby-doc.org/stdlib-2.3.0/libdoc/erb/rdoc/ERB.html), имеют встроенный механизм поддержки шаблонов. И снова следует напомнить, что при необходимости работы с шаблонами, не важно, на каком языке – Python или Go, самым важным фактором является выбор используемого языка шаблонов. Чаще всего наилучшим вариантом является механизм поддержки шаблонов, встроенный непосредственно в язык.

Универсальность шаблонов

Очень важно отметить, что концепции механизма шаблонов, особенно те, которые рассматриваются в текущей главе, в действительности не являются присущими какому-то одному варианту использования, а могут применяться практически к любому носителю информации, основанному на тексте. В этой главе мы главное внимание уделяем шаблонам для сетевых конфигураций, но шаблоны могут использоваться и для других целей, например, для создания динамических веб-страниц, как было показано в предыдущем разделе. Фактически, можно применять шаблоны и для более общих целей, например для создания оперативных отчетов. Предположим, что вы только что получили свежие данные из сетевого устройства и хотите быстро сформировать хорошо оформленный отчет, чтобы отправить его по электронной почте нескольким сотрудникам. В примере 6.1 показан пример шаблона Jinja для формирования отчета, содержащего список VLAN.

Пример 6.1 ❖ Оперативный отчет, формируемый с помощью шаблонов Jinja

```
| VLAN ID | NAME | STATUS |
| ------- |------| -------|
{% for vlan in vlans %}
| {{ vlan.get('vlan_id') }} | {{ vlan.get('name') }} | {{ vlan.get('status') }} |
{% endfor %}
```

Поскольку основная часть работы сетевого инженера является работой с текстом, можно создавать шаблоны для обработки текста. Помните об этом, когда изучаете подробности технологий шаблонов, подобных Jinja, – шаблоны очень

Tlgm: @it_boooks

хорошо работают в ограниченном наборе вариантов использования, который будет рассматриваться в текущей главе.

Важность использования шаблонов в процессе автоматизации сети

Возможно, читатели несколько удивлены тем, что в предыдущих разделах речь шла о веб-разработке, и пытаются понять, каким образом это может помочь в изучении автоматизации сети. Очень важно понимать значение шаблонов в целом, не важно, применяются ли они для веб-разработки или для решения какой-либо другой задачи. Шаблоны обеспечивают целостность и согласованность – вместо набираемых вручную текстовых файлов, содержащих огромное количество HTML-тегов, и ввода команд в текстовой строке мы можем с помощью шаблонов объявить некоторые фрагменты файлов и командных строк статическими, а другие фрагменты станут динамическими.

Каждый сетевой инженер, достаточно долго работающий с сетями, наверняка имеет в запасе заранее подготовленный файл конфигурации для нового элемента сети, такого как коммутатор или маршрутизатор. Возможно, такую работу необходимо проделать для множества коммутаторов, например при создании нового центра данных. Для рационального использования рабочего времени весьма полезно создавать соответствующие конфигурации заранее, а при необходимости физического подключения нового коммутатора в стойку или с помощью кабеля сетевой инженер просто копирует заранее заготовленную конфигурацию в командную строку терминала.

Предположим, что необходимо выполнить подобную работу – создание конфигураций для всех коммутаторов в новом центре данных вашей организации. Понятно, что для каждого коммутатора потребуется собственный специализированный файл конфигурации, но бóльшая часть конфигурационной информации будет одинаковой для всех устройств. Например, одинаковые строки группы SNMP, один и тот же пароль администратора, идентичная конфигурация VLAN (по крайней мере, для одинаковых типов устройств, например для TOR-коммутаторов). Разумеется, некоторые элементы конфигурации будут индивидуальными для различных устройств, например управляющие IP-адреса и имена хостов. Но это самые простые примеры, а как поступить с коммутатором уровня 3? Для этого устройства потребуется специализированная конфигурация подсети, возможно также, специализированные конфигурации протоколов маршрутизации, зависящие от места расположения устройства в топологии сети. Принятие решений о выборе параметров для таких коммутаторов отнимает много времени, к тому же возрастает вероятность ошибок. Шаблоны позволяют стандартизировать обобщенную базовую конфигурацию и помогают обеспечить передачу корректных значений параметров в каждое устройство. Отделение шаблона от данных, заполняющих его, – это один из наилучших способов упрощения процесса конфигурирования.

Tlgm: @it_boooks

Самое важное преимущество использования шаблонов в работе сетевых инженеров – обеспечение целостности и согласованности конфигурации. Правильно применяемые шаблоны способны реально снизить вероятность ошибок, которые могут возникать при внесении изменений вручную в конфигурации сетей. На первых порах автоматизированный способ внесения многочисленных сложных изменений в конфигурации может вызывать резкое неприятие и казаться неправильным решением, но если вы неукоснительно соблюдаете дисциплину и тщательно тестируете свои шаблоны, то сможете действительно улучшить эффективность выполняемых сетевых операций. Шаблоны не устраняют человеческие ошибки автоматически, но при правильном применении значительно снижают вероятность их возникновения и сокращают время неработоспособности сети.

Использование шаблонов для развертывания новых сетевых устройств представляет собой простой пример, подтверждающий важность шаблонов, а дополнительным преимуществом является существенная экономия рабочего времени сетевого инженера. Но это не единственная область применения шаблонов. Во многих проектах автоматизации сетей шаблоны используются даже не людьми, а программным обеспечением, выполняющим процедуры автоматизации, например Ansible, для оперативного внесения изменений в конфигурации сетевых устройств непосредственно в процессе эксплуатации сети.

В следующих разделах будут рассматриваться примеры практического применения шаблонов на языке Jinja.

Язык Jinja для создания шаблонов сетевой конфигурации

В следующих разделах подробно рассматривается один из языков шаблонов – Jinja. Сначала приводятся некоторые основные характеристики этого языка, затем мы переходим к подробностям, при этом постоянно демонстрируя, каким образом рассматриваемые концепции могут применяться для генерации логически согласованных корректных сетевых конфигураций.

Почему именно Jinja

Язык Jinja упоминался во вступительной части текущей главы, но вместе с ним упоминались и некоторые другие языки шаблонов. Почему мы выбрали именно Jinja для создания шаблонов автоматизации сети? Чтобы размер главы не увеличивался до бесконечности, мы рассматриваем только язык шаблонов Jinja, потому что он наилучшим образом сочетается со многими другими технологиями, описываемыми в книге, в том числе и с языком программирования Python (глава 4). Фактически Jinja – это язык шаблонов, созданный для Python. Кроме того, Jinja вполне подходит и активно используется в Ansible и Salt, двух инструментах автоматизации, написанных на языке Python. Оба этих инструмента будут подробно рассматриваться в главе 9. Если вы хоро-

Tlgm: @it_boooks

шо знаете Python, то сразу почувствуете и убедитесь, что язык шаблонов Jinja очень похож на него.

Разумеется, можно создавать шаблоны сетевых конфигураций и на других языках. Но если вы впервые имеете дело с языками шаблонов, особенно если следуете указаниям этой книги, чтобы приобрести некоторые навыки программирования на Python, то вскоре поймете, что Jinja является наиболее подходящим для вас инструментом автоматизации сети.

Динамическая вставка данных в простой шаблон Jinja

Начнем с простого примера – создания шаблона для конфигурации одного интерфейса коммутатора. В примере используется конфигурация реального порта коммутатора (с использованием синтаксиса командной строки в соответствии с отраслевым стандартом), которую необходимо преобразовать в шаблон (таким образом мы сможем сконфигурировать сотни других портов коммутаторов в нашей сетевой среде):

```
interface GigabitEthernet0/1
 description Server Port
 switchport access vlan 10
 switchport mode access
```

Для этого фрагмента конфигурации создать шаблон можно очень легко – нужно только решить, какие части конфигурации должны остаться неизменными, а какие должны стать динамическими. В следующем примере имя конкретного интерфейса (GigabitEthernet0/1) заменено на переменную, значение которой будет подставляться в шаблон для создания реального экземпляра конфигурации.

```
interface {{ interface_name }}
 description Server Port
 switchport access vlan 10
 switchport mode access
```

Это означает, что переменная `interface_name` передается для обработки в шаблон, и в этом месте будет вставлено значение, связанное с переменной `interface_name` в текущий момент.

Но в предыдущем примере предполагается, что для каждого сетевого интерфейса конфигурация одинакова. А если для некоторых интерфейсов требуется другая виртуальная сеть VLAN или другое описание интерфейса? В этом случае следует заменить соответствующие элементы конфигурации специализированными переменными.

```
interface {{ interface_name }}
 description {{ interface_description }}
 switchport access vlan {{ interface_vlan }}
 switchport mode access
```

Это очень простые примеры, в которых даже нет необходимости применять механизм пространств имен.

Tlgm: @it_boooks

При обработке шаблонов общепринятой практикой является применение таких концепций, как классы и словари из языков программирования, подобных Python. Это позволяет сохранять многочисленные экземпляры данных, таких как приведенные в предыдущем примере, таким образом, можно зациклить процедуру и несколько раз записывать полученную в результате конфигурацию. Циклы будут рассматриваться в одном из следующих разделов, но сейчас мы перепишем шаблон из предыдущего примера и сохраним его в файле *template.j2*, чтобы воспользоваться преимуществом, в некотором роде похожим на преимущества, присущие классу или словарю языка Python:

```
interface {{ interface.name }}
 description {{ interface.description }}
 switchport access vlan {{ interface.vlan }}
 switchport mode access
```

Это минимальное изменение, но очень важное. Объект `interface` передается в шаблон как единое целое. Если бы `interface` был классом языка Python, то `name`, `description` и `vlan` представляли бы свойства (properties) этого класса. Почти то же самое можно сказать об `interface` как о словаре, с учетом единственного различия – все перечисленные элементы являлись бы ключами словаря, а не свойствами. В обоих случаях механизм обработки шаблонов подставлял бы значения свойств или ключей автоматически.

Обработка файла шаблона Jinja средствами языка Python

В предыдущем примере рассматривался простейший шаблон Jinja для конфигурации порта коммутатора, но ничего не было сказано о том, как в действительности обрабатывается этот шаблон и какие данные подставляются в него для получения конечного результата. В текущем разделе рассматриваются эти аспекты обработки шаблонов с использованием языка Python и библиотеки Jinja2.



Сам язык обработки шаблонов называется Jinja, но библиотека языка Python для работы с шаблонами именуется Jinja2.

Воспользуемся шаблоном из предыдущего примера и применим язык Python для заполнения полей реальными данными. Этот пример будет выполняться в интерактивном интерпретаторе Python, поэтому вы можете воспроизвести его и изучить во всех подробностях на своем компьютере.



Механизм обработки шаблонов Jinja2 для языка Python не является частью стандартной библиотеки, поэтому не устанавливается по умолчанию. Библиотеку Jinja2 можно установить с помощью команды `pip install jinja2`, то есть точно так же, как и любой другой пакет Python, найденный в репозитории PyPI, в соответствии с указаниями главы 4.

После установки библиотеки Jinja2 в первую очередь необходимо импортировать требуемые объекты для обработки шаблонов.

```
>>> from jinja2 import Environment, FileSystemLoader
```

Tlgm: @it_boooks

Далее настраивается программная среда, чтобы механизм обработки шаблонов знал, где искать заданный шаблон.

```
>>> ENV = Environment(loader=FileSystemLoader('.'))
>>> template = ENV.get_template("template.j2")
```

В первой строке создается объект `Environment`, определяемый символом точки (.), означающим, что шаблоны размещены в том же каталоге, из которого был запущен интерпретатор Python. Во второй строке для этой среды порождается объект-шаблон, для которого статически определяется имя *template.j2*. Напомним, что содержимое этого файла шаблона идентично содержимому ранее сохраненного файла *template.j2*.

После этого требуются собственно данные. В этом примере используется словарь языка Python. Отметим, что ключи словаря соответствуют именам полей обрабатываемого шаблона.

```
>>> interface_dict = {
...     "name": "GigabitEthernet0/1",
...     "description": "Server Port",
...     "vlan": 10,
...     "uplink": False
... }
```



Важно помнить, что необходимость создания структур данных средствами языка Python вручную для заполнения шаблона будет возникать крайне редко. Здесь это сделано для наглядности. Но вы можете написать свое приложение для извлечения требуемых данных из других источников, чтобы не «зашивать» структуры и данные в программу обработки.

Теперь у нас есть все необходимое для обработки шаблона. Вызывается функция `render()` объекта-шаблона для передачи данных в механизм обработки, после чего используется функция `print()` для вывода результата обработки данных на экран.

```
>>> print(template.render(interface=interface_dict))
interface GigabitEthernet0/1
 description Server Port
 switchport access vlan 10
 switchport mode access
```

Отметим способ передачи аргумента в функцию `render()`. Обратите особое внимание на имя – передаваемый аргумент `interface` соответствует объекту `interface` в ранее подготовленном шаблоне Jinja. Именно так передается словарь `interface` в механизм обработки шаблонов – когда механизм обработки обнаруживает ссылки на этот объект или на его ключи, он использует переданный словарь для обращения по указанным ссылкам.

Можно видеть, что выводимый результат обработки соответствует нашим предположениям. Но совсем не обязательно всегда использовать только словарь языка Python. Это не самый часто применяемый объект для управления

Tlgm: @it_boooks

данными при использовании других библиотек Python для обработки шаблона Jinja, данные могут быть представлены в форме класса Python.

В следующем примере показана программа на языке Python, похожая на программу из предыдущего примера, но здесь вместо словаря используется класс.

```
from jinja2 import Environment, FileSystemLoader

ENV = Environment(loader=FileSystemLoader('.'))

template = ENV.get_template("template.j2")

class NetworkInterface(object):
    def __init__(self, name, description, vlan, uplink=False):
        self.name = name
        self.description = description
        self.vlan = vlan
        self.uplink = uplink

interface_obj = NetworkInterface("GigabitEthernet0/1", "Server Port", 10)

print(template.render(interface=interface_obj))
```

Эта программа выводит точно такой же результат, как и предыдущая. Поэтому в действительности не существует единственного «правильного» способа заполнения данными шаблона Jinja – все зависит от того, из какого источника берутся данные. К счастью, библиотека Jinja2 в этом отношении обеспечивает достаточную гибкость.



В этой книге классам языка Python уделяется совсем немного внимания. Причина в существовании многих других ресурсов, предназначенных для обучения практическому использованию классов в коде Python, кроме того, многие прикладные интерфейсы API, с которыми придется иметь дело сетевым инженерам, обеспечивают солидную поддержку простых структур, таких как списки или словари. Эта книга прокладывает тропинку между основами программирования и практическим применением отраслевых инструментальных средств, как ныне существующих, так и прогнозируемых в ближайшем будущем.

Тем не менее нельзя недооценивать многочисленные преимущества объектно-ориентированного программирования и использования таких сущностей, как классы. При правильном применении объектно-ориентированный код может стать более удобным для чтения, сопровождения и даже повысить эффективность тестирования. При написании собственного кода следует тщательно подумать и принять верное решение: когда необходимо определение полноценного объекта, а когда достаточно простого словаря.

Условные выражения и циклы

Теперь нужно заставить шаблоны действительно работать на нас. Предыдущие примеры предназначались для первоначального ознакомления со способами вставки динамических данных в содержимое текстового файла, но это всего лишь малая часть работы по крупномасштабному внедрению шаблонов в процесс автоматизации сетевых конфигураций.

Tlgm: @it_boooks



Jinja позволяет встраивать логику в стиле языка Python в файлы шаблонов, чтобы получить возможность принятия решений или собирать дублирующиеся данные в единый компактный фрагмент, который при обработке развертывается с помощью цикла `for`. Это весьма мощные инструменты, но их использование может стать небезопасным.

Важно не увлекаться внесением всех доступных логических конструкций в создаваемые шаблоны – Jinja предоставляет некоторые действительно полезные возможности, при этом не являясь полноценным языком программирования, – следует всегда помнить об этом и соблюдать разумный баланс.

Рекомендуем ознакомиться с документом Jinja FAQ (http://jinja.pocoo.org/docs/dev/faq/), особенно внимательно – с разделом «Isn't it a terrible idea to put Logic into Templates?» («Насколько хорош или плох подход с использованием логики в шаблонах?»), чтобы получить более подробную информацию по этой теме.

Использование условных логических конструкций для создания конфигурации порта коммутатора

Продолжим рассмотрение примера конфигурации для одного порта коммутатора, но сейчас необходимо принять решение по выбору для обработки по заданным условиям той или иной части самого файла шаблона.

Часто некоторые интерфейсы порта становятся магистральным каналом VLAN, для других определен конкретный «режим доступа». Хорошим примером является доступ к коммутатору между уровнями, где два и более интерфейсов представляют собой порты «входных соединений» (uplink), которые необходимо конфигурировать для обеспечения связи со всеми виртуальными сетями. В предыдущих примерах было показано логическое свойство uplink, для которого устанавливалось значение `True`, если интерфейс являлся «входным соединением», значение `False` – для простого порта доступа. Можно проверить значение этого свойства в шаблоне с помощью условного выражения:

```
interface {{ interface.name }}
 description {{ interface.description }}
{% if interface.uplink %}
 switchport mode trunk
{% else %}
 switchport access vlan {{ interface.vlan }}
 switchport mode access
{% endif %}
```

Итак, если для свойства `uplink` объекта `interface` установлено значение `True`, то этот интерфейс следует конфигурировать как магистральный канал VLAN. В противном случае необходимо обеспечить соответствующий режим доступа.

В предыдущем примере продемонстрирован новый синтаксис – комбинации символов `{%…%}` – специальный тег языка Jinja, обозначающий некоторый тип логической конструкции. Этот шаблон предназначен для создания конфигурации Cigabit Ethernet0/1 как магистрального канала VLAN, а любой другой интерфейс будет сконфигурирован в соответствующем режиме доступа в виртуальной сети vlan 10.

Tlgm: @it_boooks

Использование цикла для создания конфигураций нескольких портов коммутатора

Пока мы сконфигурировали только один интерфейс, но сейчас рассмотрим возможность использования циклов Jinja для создания конфигураций нескольких портов коммутатора. Для решения этой задачи воспользуемся циклом `for`, синтаксис которого очень похож на синтаксис точно такого же цикла языка Python.

```
{% for n in range(10) %}
interface GigabitEthernet0/{{ n+1 }}
 description {{ interface.description }}
 switchport access vlan {{ interface.vlan }}
 switchport mode access
{% endfor %}
```

Отметим, что здесь также используются комбинации символов {% ... %} для обозначения всех логических конструкций. В этом шаблоне вызывается функция `range()` для получения списка целых чисел, соответствующих порядковому номеру итерации, а в каждой итерации выводится результат n+1, поскольку `range()` начинает отсчет с 0, а нумерация портов коммутатора начинается с 1.

Использование цикла и условных выражений для создания различных конфигураций нескольких портов коммутаторов

В предыдущем примере создавалась одинаковая конфигурация для 10 портов коммутатора, но что делать, если для нескольких портов требуются различные конфигурации? Вернемся к ранее расмотренному примеру, в котором изучались условные выражения Jinja, с предположением, что первый порт является магистральным каналом VLAN. Для решения поставленной задачи объединим все полученные к настоящему моменту знания об условных выражениях и циклах:

```
{% for n in range(10) %}

interface GigabitEthernet0/{{ n+1 }}
 description {{ interface.description }}
{% if n+1 == 1 %}
 switchport mode trunk
{% else %}
 switchport access vlan {{ interface.vlan }}
 switchport mode access
{% endif %}

{% endfor %}
```

Здесь GigabitEthernet0/1 конфигурируется как магистральный канал VLAN, но для GigabitEthernet0/2-10 устанавливается соответствующий режим доступа. Ниже приведен пример результатов обработки имитационных данных для описаний интерфейсов:

```
interface GigabitEthernet0/1
 description TRUNK INTERFACE
```

Tlgm: @it_boooks


```
 switchport mode trunk
interface GigabitEthernet0/2
 description ACCESS INTERFACE
 switchport mode access
interface GigabitEthernet0/3
 description ACCESS INTERFACE
switchport mode access
...
```

Использование составных переменных в цикле for для генерации конфигураций

В предыдущих примерах уже применялась методика обращения к ключам для доступа к значениям словаря в шаблонах Jinja, но часто возникает необходимость последовательного прохода по значениям таких объектов, как словари и списки с применением цикла `for`.

Предположим, что в шаблон передается показанный ниже список, сохраняемый в переменной `interface_list`. В коде Python это выглядит так:

```
intlist = [
    "GigabitEthernet0/1",
    "GigabitEthernet0/2",
    "GigabitEthernet0/3"
]
print(template.render(interface_list=intlist))
```

К списку `interface_list` нужно организовать обращение в цикле, то есть необходимы возможность доступа к каждому элементу списка и генерация индивидуальной конфигурации для каждого порта коммутатора. Отметим, что здесь также изменены вложенные условные выражения, поскольку переменная-счетчик `n` в этом примере уже не используется.

```
{% for iface in interface_list %}

interface {{ iface }}
{% if iface == "GigabitEthernet0/1" %}
 switchport mode trunk
{% else %}
 switchport access vlan 10
 switchport mode access
{% endif %}

{% endfor %}
```

В этом примере мы просто обращаемся к объекту `iface` для извлечения текущего элемента списка на каждой итерации цикла.

Эту задачу можно решить с помощью словаря. Сначала приведем соответствующий фрагмент кода Python для создания и передачи требуемого словаря в наш шаблон Jinja. Для упрощения первой версии примера передается только набор имен интерфейсов в качестве ключей, а значениями являются соответствующие описания портов.

Tlgm: @it_boooks


```
intdict = {
    "GigabitEthernet0/1": "Server port number one",
    "GigabitEthernet0/2": "Server port number two",
    "GigabitEthernet0/3": "Server port number three"
}
print(template.render(interface_dict=intdict))
```

Изменить цикл для последовательного прохода по созданному словарю можно почти таким же образом, как это делается в коде Python:

```
{% for name, desc in interface_dict.items() %}
interface {{ name }}
  description {{ desc }}
{% endfor %}
```

Выражение `for name, desc...` означает, что на каждой итерации цикла переменная `name` будет содержать ключ подготовленного словаря, а переменная `desc` – соответствующее значение для этого ключа. Не забудьте добавить встроенный метод `items()` для правильного извлечения ключей и значений.

Такой подход позволяет просто обращаться к переменным `name` и `desc` в теле шаблона, а результат его обработки показан ниже:

```
interface GigabitEthernet0/3
  description Server port number three
interface GigabitEthernet0/2
  description Server port number two
interface GigabitEthernet0/1
  description Server port number one
```



Возможно, вы заметили, что интерфейсы здесь выведены в том же порядке, что и в предыдущем примере. Причина в том, что словари не упорядочены, и проход по их элементам выполняется с помощью цикла `for`. Вероятно, вы помните, что это свойство проявлялось, когда мы изучали словари в главе 4.

Вы наверняка обратили внимание на некоторые ограничения в показанных выше примерах. В некоторых примерах итерации выполнялись с помощью функции `range()`, то есть не извлекались все значимые метаданные об интерфейсах, которые становились доступными при использовании классов или словарей. Но даже при использовании словаря в последнем примере его структура содержала лишь имя интерфейса и его краткое описание.

Генерация конфигураций интерфейсов из списка словарей

В завершающем примере будет продемонстрировано объединенное использование списков и словарей для создания шаблона, наиболее соответствующего реальным рабочим критериям. Для каждого интерфейса будет сформирован собственный словарь, в котором ключами являются стандартные атрибуты любого сетевого интерфейса, такие как `name`, `description` или `uplink`. Все словари будут храниться в общем списке, который будет обрабатывать шаблон, чтобы на каждой итерации получить соответствующую конфигурацию.

Tlgm: @it_boooks

Сначала формируется описанная выше структура данных с помощью кода Python:

```
interfaces = [
    {
        "name": "GigabitEthernet0/1",
        "desc": "uplink port",
        "uplink": True
    },
    {
        "name": "GigabitEthernet0/2",
        "desc": "Server port number one",
        "vlan": 10
    },
    {
        "name": "GigabitEthernet0/3",
        "desc": "Server port number two",
        "vlan": 10
    }
]
print(template.render(interface_list=interfaces))
```

Такая структура позволяет написать чрезвычайно эффективный шаблон, в котором выполняется последовательный проход по предложенному списку и для каждого элемента этого списка обеспечивается обращение к ключам, обнаруженным в текущем экземпляре словаря. Следующий пример демонстрирует практическое применение всех методик, о которых мы узнали, изучая циклы и условные выражения.

```
{% for interface in interface_list %}
interface {{ interface.name }}
  description {{ interface.desc }}
  {% if interface.uplink %}
    switchport mode trunk
  {% else %}
    switchport access vlan {{ interface.vlan }}
    switchport mode access
  {% endif %}
{% endfor %}
```



При доступе к данным в словарях с использованием языка шаблонов Jinja можно применять обычный синтаксис языка Python, то есть выражения типа `dict['key']` или их сокращенную форму `dict.key`. Оба выражения абсолютно равнозначны, и если вы пытаетесь обратиться к отсутствующему ключу, то возникает ошибка доступа (key error). Но в языке шаблонов Jinja можно также воспользоваться методом `get()`, если ключ не является обязательным (то есть может отсутствовать) или при отсутствии ключа необходимо вернуть некоторое конкретное значение, например `dict.get(key, 'UNKNOWN')`.

Как было сказано ранее, встраивание данных непосредственно в приложения Python представляет собой неэффективную практику программирования

Tlgm: @it_boooks

и поэтому не рекомендуется (см. список словарей с именем `interfaces` в предыдущем примере). Вместо этого разместим наши данные в отдельном YAML-файле и соответствующим образом перепишем приложение, чтобы обеспечить импортирование требуемых данных с последующей обработкой их в шаблоне. Это весьма эффективная, рекомендуемая практика, поскольку она позволяет даже пользователям без опыта работы с языком Python редактировать сетевую конфигурацию, внося изменения в простой YAML-файл.

Ниже приведен пример YAML-файла, содержащего список интерфейсов, аналогичный структуре из примера предыдущего раздела.

```
---
- name: GigabitEthernet0/1
  desc: uplink port
  uplink: true
- name: GigabitEthernet0/2
  desc: Server port number one
  vlan: 10
- name: GigabitEthernet0/3
  desc: Server port number two
  vlan: 10
```

В главе 5 было показано, что импортирование YAML-файла средствами языка Python выполняется очень просто. В качестве напоминания приведем полный код приложения на языке Python, но вместо статического встроенного списка словарей просто импортируется содержимое YAML-файла для извлечения данных.

```
from jinja2 import Environment, FileSystemLoader
import yaml

ENV = Environment(loader=FileSystemLoader('.'))

template = ENV.get_template("template.j2")

with open("data.yml") as f:
    interfaces = yaml.load(f)
    print(template.render(interface_list=interfaces))
```

Можно воспользоваться ранее созданным шаблоном и получить точно такой же результат, но на этот раз данные для обработки в шаблоне берутся из внешнего YAML-файла, сопровождение которого не вызовет никаких затруднений у пользователя с любым уровнем подготовки. Теперь файл с кодом Python содержит только логику извлечения данных и вызов шаблона для их обработки. В совокупности это формирует более удобную для сопровождения систему обработки шаблонов.

В этом разделе мы рассмотрели лишь небольшую часть основных возможностей циклов и условных выражений. Предлагаем читателю самостоятельно продолжить изучение этих концепций и активно применять их в своей практической работе.

Tlgm: @it_boooks

Фильтры Jinja

Иногда необходимо применить некоторый тип обработки к некоторой переменной внутри шаблона. Простым примером может быть преобразование букв небольшого фрагмента текста в верхний регистр.

Эту задачу позволяют решить фильтры. Подобно тому, как в командной оболочке (shell) ОС Linux можно в конвейере передавать вывод одной команды на ввод другой, результат команд Jinja также можно перенаправлять в фильтр. Полученный в результате текст выводится как результат обработки в шаблоне.

Использование фильтра upper

Снова обратимся к последнему примеру шаблона, но теперь используем встроенный фильтр для преобразования в верхний регистр всех букв описаний в конфигурации каждого интерфейса.

```
{% for interface in interface_list %}
interface {{ interface.name }}
  description {{ interface.desc|upper }}
  {% if interface.uplink %}
    switchport mode trunk
  {% else %}
    switchport access vlan {{ interface.vlan }}
    switchport mode access
  {% endif %}
{% endfor %}
```

После имени переменной `interface.desc`, но обязательно внутри фигурных скобок можно использовать символ конвейера (pipe) (|) для передачи данных из переменной `desc` в фильтр `upper`. Этот фильтр встроен в библиотеку Jinja2 языка Python и позволяет выполнить преобразование букв в верхний регистр в переданном тексте.

Объединение фильтров в цепочку

Можно также соединять фильтры друг с другом, практически так же, как объединяются в конвейер команды в ОС Linux или вызовы методов в коде Python. Воспользуемся фильтром `reverse`, чтобы принять текст в верхнем регистре и вывести его в обратном порядке (от конца к началу).

```
{% for interface in interface_list %}
interface {{ interface.name }}
  description {{ interface.desc|upper|reverse }}
  {% if interface.uplink %}
    switchport mode trunk
  {% else %}
    switchport access vlan {{ interface.vlan }}
    switchport mode access
  {% endif %}
{% endfor %}
```

Tlgm: @it_boooks

Обработка с помощью этого шаблона дает следующий результат:

```
interface GigabitEthernet0/1
  description TROP KNILPU
  switchport mode trunk
interface GigabitEthernet0/2
  description ENO REBMUN TROP REVRES
  switchport access vlan 10
  switchport mode access
interface GigabitEthernet0/3
  description OWT REBMUN TROP REVRES
  switchport access vlan 10
  switchport mode access
```

Исходное описание GigabitEthernet0/1 представляло собой строку «uplink port», затем с помощью фильтра `upper` выполняется преобразование этой строки в верхний регистр, то есть «UPLINK PORT», потом фильтр `reverse` меняет порядок букв в строке на противоположный – «TROP KNILPU», после чего окончательный результат обработки выводится на экран терминала.

Создание пользовательских фильтров для Jinja

В документации языка шаблонов Jinja подробно описаны многочисленные встроенные фильтры, весьма полезные для пользователей. Но иногда необходим собственный специализированный фильтр. В некоторых особенных вариантах автоматизации сети требуется выполнение специфических операций фильтрации, которые могут не поддерживаться библиотекой Jinja2.

К счастью, библиотека позволяет создавать собственные фильтры. В следующем примере показан полный скрипт на языке Python, в котором определяется новая функция `get_interface_speed()`. Эта функция проста – она ищет ключевые слова, такие как «gigabit» или «fast», в переданном строковом аргументе и возвращает соответствующее числовое значение скорости в Mbps (Мбит/с). Здесь данные также загружаются из внешнего YAML-файла, продолжая применение подхода, принятого в предыдущих примерах.

```
# Импорт библиотек Jinja2 и PyYAML
from jinja2 import Environment, FileSystemLoader
import yaml

# Объявление среды выполнения шаблона
ENV = Environment(loader=FileSystemLoader('.'))

def get_interface_speed(interface_name):
    """ get_interface_speed returns the default Mbps value for a given
        network interface by looking for certain keywords in the name
    """
    if 'gigabit' in interface_name.lower():
        return 1000
    if 'fast' in interface_name.lower():
        return 100
```

Tlgm: @it_boooks


```
# Фильтры добавляются в объект ENV после его объявления. Отметим, что
# в действительности здесь передается только имя функции get_interface_speed,
# но она не вызывается - механизм шаблонов автоматически выполнит эту функцию
# после вызова template.render().
ENV.filters['get_interface_speed'] = get_interface_speed
template = ENV.get_template("templatestuff/template.j2")

# Загружаются данные из YAML-файла и передаются в шаблон для обработки.
with open("templatestuff/data.yml") as f:
    interfaces = yaml.load(f)
    print(template.render(interface_list=interfaces))
```

Немного изменив наш шаблон, как показано в следующем примере, можно применить созданный фильтр, то есть передать аргумент `interface.name` в функцию-фильтр `get_interface_speed`. Выводимым результатом будет некоторое целочисленное значение, которое функция-фильтр выберет по заданным условиям и вернет. Поскольку все имена интерфейсов GigabitEthernet, выводится значение скорости 1000.

```
{% for interface in interface_list %}
interface {{ interface.name }}
  description {{ interface.desc|upper|reverse }}
  {% if interface.uplink %}
    switchport mode trunk
  {% else %}
    switchport access vlan {{ interface.vlan }}
    switchport mode access
  {% endif %}
  speed {{ interface.name|get_interface_speed }}
{% endfor %}
```

Использование существующего кода Python как фильтра Jinja

Для создания специализированного фильтра Jinja не всегда обязательно писать код своей функции. Иногда можно подобрать существующую функцию Python, которая будет прекрасно работать в качестве фильтра. Воспользоваться функцией языка Python достаточно просто – нужно правильно импортировать ее и передать в шаблон точно так же, как если бы она была написана вами.

Для примера возьмем функцию из библиотеки `bracket_expansion` для быстрого формирования набора имен интерфейсов без создания списка или словаря вручную. Чтобы лучше понять, как работает предложенная методика, обратите внимание на комментарии в коде примера.

```
# Импортирование библиотеки Jinja2
from jinja2 import Environment, FileSystemLoader

# bracket_expansion - библиотека от независимого производителя.
# Чтобы воспользоваться этой библиотекой, сначала установите ее с помощью pip.
from bracket_expansion import bracket_expansion

# Объявление среды выполнения шаблона
ENV = Environment(loader=FileSystemLoader('.'))
```

Tlgm: @it_boooks


```
# Фильтры добавляются в объект ENV после его объявления.
# bracket_expansion - функция, передаваемая в механизм шаблонов;
# она будет автоматически выполняться при обработке шаблона.
ENV.filters['bracket_expansion'] = bracket_expansion
template = ENV.get_template("template.j2")

# Функция bracket_expansion, передаваемая в качестве фильтра, требует
# текстового образца для работы с ним. Такой образец передается
# в переменной iface_pattern
print(template.render(iface_pattern='GigabitEthernet0/0/[0-3]'))
```

Создание специализированных фильтров – весьма мощный инструмент, и вы, разумеется, могли бы написать свою функцию на языке Python, чтобы передать ее на следующий уровень обработки. Экспериментируйте с другими функциями и библиотеками обработки текста или пишите собственные инструментальные средства обработки.

Наследование шаблонов в языке Jinja

При создании более крупных и мощных шаблонов для конкретной сетевой конфигурации может потребоваться возможность разделения шаблонов на более мелкие компоненты, специализирующиеся на выполнении одной определенной задачи. Обычно создаются отдельные шаблоны для конфигурации VLAN, для интерфейсов и, возможно, для протокола маршрутизации. Такой подход к организации инструментальных средств не обязателен, но может обеспечить гораздо большую гибкость. При этом возникает вопрос: как правильно объединить все эти шаблоны, чтобы сформировать полную сетевую конфигурацию?

Язык шаблонов Jinja позволяет выполнять операцию наследования (inheritance) в файле шаблона, обеспечивая вполне успешное решение этой задачи. Например, если имеется файл *vlans.j2*, содержащий только конфигурацию VLAN, то можно выполнить процедуру наследования этого файла, чтобы получить конфигурацию VLAN в другом файле шаблона. Можно написать шаблон для конфигурации интерфейса, в который желательно включить конфигурацию VLAN из другого шаблона. В следующем примере показано, как это делается с помощью команды `include`:

```
{% include 'vlans.j2' %}

{% for name, desc in interface_dict.items() %}
interface {{ name }}
  description {{ desc }}
{% endfor %}
```

Здесь будет обработано содержимое файла `vlans.j2`, а полученный в результате текст будет передан для обработки в шаблон, в который включен файл `vlans.j2`. При помощи команды `include` авторы шаблонов могут компоновать конфигурации коммутаторов из различных составных частей (модулей). Это великолепный инструмент, позволяющий поддерживать порядок и организованность.

Другим инструментом наследования Jinja является команда `block`. Это мощный, но более сложный метод наследования, который в некоторой степени по-

Tlgm: @it_boooks

хож на наследование объектов в более формализованных языках программирования, таких как Python. Используя блоки, можно определять части шаблона, которые могут быть замещены шаблоном-потомком, если такой имеется в наличии. Если шаблон-потомок отсутствует, в исходном шаблоне остается текст, заданный по умолчанию.

Следующий пример демонстрирует включение блока с помощью команды `block` в текст шаблона-родителя:

```
{% for interface in interface_list %}
interface {{ interface.name }}
  description {{ interface.desc }}
{% endfor %}
!
{% block http %}
  no ip http server
  no ip http secure-server
{% endblock %}
```

Этот шаблон сохранен в файле *no-http.j2*, при этом имя подсказывает, что в обычной ситуации встроенный HTTP-сервер данного коммутатора отключается. Но есть возможность использования блоков для обеспечения большей гибкости. Можно создать шаблон-потомок *yes-http.j2*, предназначенный для замещения исходного блока, чтобы получить конфигурацию, разрешающую работу встроенного HTTP-сервера, если это необходимо.

```
{% extends "no-http.j2" %}
{% block http %}
  ip http server
  ip http secure-server
{% endblock %}
```

Это позволяет разрешить использование HTTP-сервера непосредственно при обработке шаблона-потомка. В первой строке предыдущего примера выполняется «расширение» (командой `extends`) функциональности родительского шаблона *no-http.j2*, для того чтобы все конфигурации интерфейсов непременно включались в общий результат обработки. Но поскольку обрабатывается шаблон-потомок, его блок http замещает указанный фрагмент родительского шаблона. Использование блоков таким способом очень удобно для изменения отдельных частей конфигурации, но такой подход не в полной мере обеспечивается обычной заменой переменных.

Для получения более подробной информации о механизме наследования шаблонов Jinja обратитесь к официальной документации (http://jinja.pocoo.org/docs/dev/templates/#template-inheritance), которую следует всегда держать под рукой.

Создание переменных в Jinja

Язык шаблонов Jinja позволяет создавать переменные в самом шаблоне. Это делается с помощью команды `set`. Самый распространенный вариант – создание

Tlgm: @it_boooks

переменной с коротким именем. Иногда необходимо работать с несколькими вложенными словарями или объектами языка Python, чтобы получить доступ к требуемым данным, которые будут использоваться многократно в шаблоне. Чтобы не повторять несколько раз длинную строку свойств или ключей, применяется команда `set` для представления некоторого значения более коротким именем переменной:

```
{% set int_desc = switch01.config.interfaces['GigabitEthernet0/1']['description'] %}
{{ int_desc }}
```

Резюме

После вводного курса Jinja важно охватить все изученное более обобщенным взглядом и понять, где могут или должны применяться шаблоны в контексте сетевой автоматизации. При изучении примеров этой главы может сложиться впечатление, что для использования шаблонов необходимо писать код на языке Python.

В определенной степени так и есть, поскольку это весьма эффективная методика использования шаблонов для управления сетевой конфигурацией, но это далеко не единственная возможность. В главе 9 мы рассмотрим инструменты Ansible и Salt, позволяющие определять данные конфигурации в простом формате YAML и вставлять эти данные в шаблон без необходимости написания какого-либо программного кода. Если шаблон достаточно прост и требуется методика создания шаблонов из существующей конфигурации, то это самый простой способ генерации шаблонов.

Приведем несколько итоговых обобщенных правил использования шаблонов для автоматизации сети:

- создавайте как можно более простые шаблоны. Циклы и условные выражения улучшают логику шаблонов, но не следует злоупотреблять ими. Jinja не настолько универсален, как полноценные языки программирования типа Python, поэтому рекомендуем выносить более сложную логику за пределы шаблонов;
- применяйте механизм наследования шаблонов для многократного использования фрагментов конфигураций без дублирования кода;
- помните о том, что код шаблона и данные должны обрабатываться отдельно. Например, следует хранить идентификаторы VLAN ID в отдельном файле данных (возможно, в формате YAML), а синтаксические конструкции командной строки (CLI) для создания конфигураций VLAN необходимо разместить в специализированном шаблоне;
- используйте систему управления версиями (например, Git) для хранения всех вариантов своих шаблонов.

Следуя этим простым правилам, вы сможете эффективно применять шаблоны в повседневной работе по автоматизации сети.

Tlgm: @it_boooks

Глава 7

Использование сетевых прикладных программных интерфейсов (API)

Изучая основы программирования на языке Python и форматы данных, затем создание шаблонов конфигураций на языке Jinja, мы осваивали главные основополагающие технологии и практические методики, которые помогут вам стать более профессиональным сетевым инженером. В этой главе мы рассмотрим практическое применение описанных ранее методик, а также начнем изучение способов взаимодействия с различными типами прикладных программных интерфейсов (API) сетевых устройств.

Чтобы помочь вам лучше понять, как приступить к автоматизации сетей на практике, глава разделена на три части:

- основы сетевых API – подробно рассматривается архитектура и основные характеристики различных API, в том числе RESTful API на основе протокола HTTP, API, не относящиеся к категории RESTful, на основе протокола HTTP, а также NETCONF;
- практическое применение сетевых API – рассматриваются инструментальные средства, наиболее часто используемые для тестирования и исследования практического применения каждого типа API;
- автоматизация с использованием сетевых API – рассматриваются библиотеки Python, позволяющие организовать процесс автоматизации сетей. Приводится описание библиотеки requests на языке Python для применения API на основе протокола HTTP, библиотеки ncclient для взаимодействия с устройствами NETCONF, а также библиотеки netmiko для автоматизации сетевых устройств на основе протокола SSH.

Но при изучении главы следует помнить о том, что излагаемый в ней материал не является исчерпывающим руководством по каждому конкретному API и не может заменить официальную документацию. В главе приводятся примеры, использующие собственные реализации того или иного API от различных

Tlgm: @it_boooks

производителей сетевого оборудования, поскольку весьма часто приходится работать в сетевой среде, объединяющей устройства от многих производителей. Кроме того, важно хорошо знать общие варианты применения и существенные различия между разнообразными реализациями одного и того же типа API.

Основы сетевых API

В этом разделе главное внимание уделено двум наиболее широко распространенным типам API для сетевых устройств: API на основе протокола HTTP и API на основе NETCONF. Сначала будут рассматриваться основополагающие концепции каждого типа API, после чего мы перейдем к практическому применению API с примерами использования конкретных реализаций различными производителями сетевого оборудования.

Начнем с изучения RESTful API на основе протокола HTTP.

Введение в API-интерфейсы на основе протокола HTTP

Существует два типа API на основе протокола HTTP, которые необходимо хорошо знать в качестве сетевых API: RESTful API на основе протокола HTTP и API на основе протокола HTTP, не принадлежащие к категории RESTful. Чтобы лучше понять их особенности и смысл термина RESTful, начнем изучение с RESTful API. Когда вы освоите архитектуру и основные концепции стиля RESTful, мы сравним подход, принятый в RESTful, с API на основе протокола HTTP, не использующими концепции RESTful.

RESTful API

Прикладные программные интерфейсы RESTful API становятся все более распространенными и широко используются в сетевой отрасли, несмотря на то что они появились лишь в начале 2000 гг. Большинство API, существующих в наши дни в сетевой инфраструктуре, представляют собой RESTful API на основе протокола HTTP. Когда вы слышите о RESTful API на сетевом устройстве или на контроллере SDN, это означает, что такой API обеспечивает обмен информацией между клиентом и сервером.

Клиент обычно представлен приложением, таким как скрипт на языке Python или веб-приложением с пользовательским интерфейсом, а сервером является сетевое устройство или контроллер. Поскольку протокол HTTP используется как транспортный протокол, пользователь должен выполнять некоторые операции с указанием URL, точно так же, как это делается при перемещениях по интернету (World Wide Web). Таким образом, если вы знаете, что при обращении к какому-либо веб-сайту выполняется операция HTTP GET, а при заполнении любой веб-формы после щелчка по кнопке **Submit** выполняется операция HTTP POST, вы уже понимаете основы работы с RESTful API.

Рассмотрим примеры получения данных с веб-сайта и получения данных из сетевого устройства с помощью RESTful API. В обоих случаях запрос HTTP GET отправляется на веб-сервер (см. рис. 7.1).

Tlgm: @it_boooks

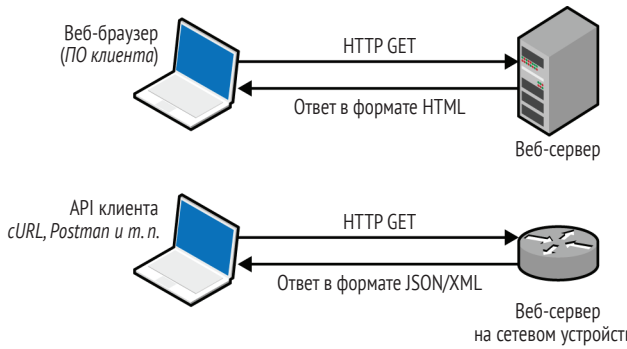


Рис. 7.1 ❖ Представление концепций REST на примере ответов на запрос HTTP GET

На рис. 7.1 одним из главных различий являются данные, передаваемые на веб-сервер и обратно от веб-сервера. В интернете вы получаете данные в формате HTML, которые интерпретируются браузером для правильного отображения внешнего вида и содержимого веб-сайта. Если же запрос HTTP GET обращен к веб-серверу, поддерживающему RESTful API (напомним: доступ к такому веб-сайту осуществляется через URL), то возвращаемые данные, как правило, кодируются в формате JSON или XML. Эти форматы рассматривались в главе 5, и мы будем пользоваться ими в дальнейшем. Поскольку данные возвращаются в формате JSON/XML, приложение клиента обязано знать, как интерпретируется формат JSON и/или XML. Поэтому перед началом изучения практического применения RESTful HTTP API необходимо получить более полное представление об этих концепциях.

После ознакомления с работой RESTful API на самом высоком уровне заглянем немного глубже и рассмотрим базовые принципы функционирования RESTful API. Следует особо отметить, что первоначально основы и структура современных RESTful API, функционирующих в веб-среде, были изложены в кандидатской диссертации (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) Роя Филдинга (Roy Fielding) в 2000 году. В диссертации под названием «Architectural Styles and the Design of Network-based Sotware Architectures» Филдинг определил сложные подробности работы с сетевыми системами в сети интернет, использующими архитектуру, обозначенную специальным термином REST (REpresentational State Transfer – передача состояния представления).

Существуют шесть архитектурных ограничений (или условий), которым непременно должен соответствовать интерфейс, чтобы считаться совместимым с REST или RESTful. Для наших текущих целей достаточно расмотреть три из этих шести условий:

- *клиент-сервер* – требование усовершенствования эффективности использования систем при одновременном упрощении требований к сер-

Tlgm: @it_boooks

веру. Архитектура клиент-серевер позволяет обеспечить переносимость и взаимозаменяемость клиентских приложений без необходимости замены компонентов сервера. Это означает, что можно использовать различные клиентские API (пользовательский веб-интерфейс, интерфейс командной строки), потребляющие некоторые ресурсы сервера (внутренний API);

- *отсутствие сохранения состояния (stateless)* – обмен информацией между клиентом и сервером должен осуществляться без сохранения состояния. Клиенты, применяющие формы передачи данных без сохранения состояния, должны передавать все данные, необходимые серверу для понимания и выполнения затребованной операции, в одном запросе. В качестве противоположного примера можно привести интерфейсы, подобные SSH, которые поддерживают постоянное соединение между клиентом и сервером;
- *универсальный интерфейс* – отдельные ресурсы, включаемые в область видимости (доступности) каждого конкретного вызова API, идентифицируются в сообщениях-запросах HTTP. Например, в RESTful-системах на основе HTTP указываемый URL обычно ссылается на конкретный ресурс. В сетевом контексте ресурсы соответствуют характеристикам сетевого устройства, таким как имя хоста, интерфейс, конфигурация протокола маршрутизации или любой другой ресурс, существующий на указанном устройстве. Универсальный интерфейс также определяет, что клиент должен обладать информацией о ресурсе, достаточной для создания, изменения или удаления этого ресурса.

Это всего лишь три ограничения, присущих архитектуре REST, но мы уже наблюдали сходство между функционированием RESTful-систем и повседневным использованием интернета с помощью веб-браузера. Следует помнить о том, что протокол HTTP является основным средством реализации RESTful API, хотя теоретически тип транспортного протокола может быть каким-либо другим. Чтобы лучше понять RESTful API, необходимо хорошо знать основы HTTP.

Типы запросов HTTP Важно хорошо понимать, что каждый рассматриваемый здесь RESTful API работает на основе HTTP, тем не менее не все API на основе HTTP соблюдают принципы REST, следовательно, не все являются RESTful API. Но в любом случае работа с этими API требует понимания протокола HTTP. Поскольку HTTP используется как транспортный протокол, мы будем работать с теми же типами запросов и кодами ответов, которые постоянно используются в интернете.

Например, наиболее часто применяемыми типами HTTP-запросов являются GET, POST, PATCH, PUT и DELETE. Из названий можно понять, что запросы GET используются для получения данных от сервера, запросы DELETE удаляют некоторый ресурс на сервере, а остальные три запроса POST, PATCH, PUT применяются для внесения изменений на сервере.

Tlgm: @it_boooks

При использовании в сетевой среде можно интерпретировать перечисленные типы запросов следующим образом:

- GET – получение конфигурации или текущих эксплуатационных данных;
- PUT, POST, PATCH – внесение изменений в конфигурацию;
- DELETE – удаление заданной конфигурации.

В табл. 7.1 приведены характеристики этих типов запросов. Немного позже в текущей главе мы рассмотрим каждый тип запроса на практических примерах.

Таблица 7.1. Типы запросов HTTP

Тип запроса	Описание
GET	Извлечение данных заданного ресурса
PUT	Создание или замена ресурса
PATCH	Создание или обновление объекта ресурса
POST	Создание объекта ресурса
DELETE	Удаление заданного ресурса

Коды ответов HTTP Как и типы запросов, коды ответов HTTP в RESTful API полностью совпадают с кодами ответов, используемыми при работе с веб-браузером в интернете.

Вероятно, вам приходилось видеть сообщение «401 Unauthorized» после ввода некорректного пароля и/или имени при попытке войти на веб-сайт. Вы получите точно такой же код ответа, если попытаетесь войти в систему, использующую RESTful API, и введете некорректные данные. То же самое можно сказать и о сообщениях об успешном завершении запрошенных операций, а также о внутренних ошибках сервера. В табл. 7.2 показан список общих типов кодов ответов, которые вы можете увидеть при работе с API на основе протокола HTTP. Разумеется, этот список нельзя назвать полным, в других источниках вы найдете более подробные списки кодов ответа.

Таблица 7.2. Коды ответов HTTP

Код ответа	Описание
2XX	Успешное выполнение операции по запросу
4XX	Ошибка на стороне клиента
5XX	Ошибка на стороне сервера

Итак, мы выяснили, что типы кодов ответов для API на основе протокола HTTP ничем не отличаются от стандартных кодов ответов HTTP. Здесь намеренно приведена таблица групп кодов ответов, чтобы оставить читателю возможность самостоятельного изучения конкретных кодов ответов в качестве «домашнего задания».

Tlgm: @it_boooks

После ознакомления с основополагающими принципами REST и HTTP важно уделить немного внимания API на основе протокола HTTP, не использующим концепции RESTful.

API на основе протокола HTTP, не использующие концепции RESTful

Несмотря на то что RESTful API являются более распространенными, на практике вы также можете встретиться с API на основе протокола HTTP, не использующими концепции RESTful. В сетевой отрасли чаще всего используются API, основанные на интерфейсе командной строки (CLI), то есть в действительности через API текстовые команды передаются в устройство, вместо того чтобы передавать необходимые структурированные данные. Наиболее предпочтительный подход – наличие любого современного веб-интерфейса или CLI для сетевой платформы, использующего внутренний API более низкого уровня, но для «унаследованных из прошлого», давно существующих систем, в которые встроены текстовые команды, чаще наблюдается использование API, несовместимых с REST, потому что проще добавить более подходящий API, нежели менять всю архитектуру сопровождаемой системы.

Между RESTful API и API, несовместимыми с REST, можно определить два основных различия. В предыдущем разделе была представлена концепция типов запросов протокола HTTP, обозначаемых соответствующими английскими глаголами: GET, POST, PATCH, PUT и DELETE. RESTful API используют те же глаголы для требования типа изменений, вносимых на целевом сервере. Например, в сетевом контексте изменение конфигурации никогда не произойдет, если выполняется запрос HTTP GET, потому что при этом запросе данные только возвращаются в источник запроса. Но системы, работающие по протоколу HTTP, при этом не соблюдающие принципы REST, могут применять один и тот же глагол-запрос HTTP для каждого обращения к API. Таким образом, если данные извлекаются или вносятся изменения в конфигурацию, то все обращения к API могут ограничиться лишь запросом POST. Если только этот запрос наблюдается в рассматриваемом типе API, то это API на основе протокола HTTP, но не RESTful API.

Другим существенным различием является особая значимость URL, используемого в отдельных вызовах API. Если при работе с API на основе протокола HTTP всегда указывается один и тот же URL, при этом пользователю не разрешается доступ к какому-либо конкретному ресурсу посредством изменения URL, то вы имеете дело с RESTful API на основе протокола HTTP.

Применение API на основе протокола HTTP в сетевой инфраструктуре – это большой шаг в правильном направлении для всей отрасли, но в идеальном варианте все API на основе HTTP должны соблюдать принципы REST.

При использовании и RESTful и не-RESTful API на основе протокола HTTP используются одни и те же инструментальные средства, но при этом важно хорошо понимать описанные выше различия сетевых API, поскольку API, не

Tlgm: @it_boooks

соблюдающие принципов REST, обычно не способны обеспечить такую гибкость, как RESTful API.

Теперь вам известны основные концепции работы API на основе протокола HTTP, и можно перейти к рассмотрению NETCONF API.

Основы NETCONF

NETCONF – это протокол управления сетью, определенный в документе RFC 6241 (https://tools.ietf.org/html/rfc6241) и предназначенный исключительно для управления конфигурацией, а также для получения данных конфигурации и данных о текущем эксплуатационном состоянии из сетевых устройств. В этом отношении NETCONF четко разграничивает конфигурацию и эксплуатационное (рабочее) состояние. Запросы API используются для выполнения различных операций, таких как получение состояния конфигурации, получение текущего эксплуатационного состояния и внесение изменений в конфигурацию.



В предыдущем разделе отмечалось, что RESTful API не являются новейшим достижением, но для сетевых устройств и контроллеров SDN это несомненно нововведение. Поскольку мы начинаем изучение NETCONF API, следует отметить, что NETCONF также не является новшеством. Стек протоколов NETCONF существует уже более десяти лет. В действительности первый документ RFC, определяющий отраслевой стандартный протокол, был написан в 2005 году. В течение нескольких лет он применялся на различных сетевых устройствах, но чаще всего как редко используемый API с ограниченными возможностями.

Одной из ключевых характеристик NETCONF является его способность одновременно использовать различные хранилища данных конфигурации. Большинство сетевых инженеров знакомо с понятиями текущих рабочих конфигураций (running configurations) и первоначальных конфигураций (startup configurations). Их можно мысленно представить в виде двух файлов конфигурации, но с точки зрения NETCONF – это два хранилища данных конфигурации. В реализациях NETCONF часто встречается третий вариант хранилища данных, называемый конфигурацией-кандидатом (candidate configuration). Хранилище данных конфигурации-кандидата содержит объекты конфигурации (команды CLI, если для конфигурации используется CLI-интерфейс), которые пока еще не применены к устройству. Например, если конфигурация вводится на устройстве, которое поддерживает конфигурации-кандидаты, то вводимые изменения не начинают действовать немедленно. Вместо этого предложенные изменения сохраняются в конфигурации-кандидате и только тогда применяются на устройстве, когда выполнена операция приема-подтверждения (commit). После завершения операции commit конфигурация-кандидат записывается как текущая рабочая конфигурация.

Концепция хранилища данных конфигурации-кандидата существует уже давно, поскольку изначально была определена в первом документе NETCONF RFC более десяти лет назад. Одна из проблем сетевой отрасли заключалась

Tlgm: @it_boooks

в создании работоспособных реализаций NETCONF, способных обеспечить эту функциональную особенность. Но не все попытки были совершенно бесполезными, создавались и достаточно успешные реализации. Сначала компания Juniper предложила Junos как надежную реализацию NETCONF с поддержкой конфигурации-кандидата, используемую в течение нескольких лет. Позднее компания Cisco добавила в IOS-XR расширенную поддержку протокола NETCONF, в том числе и поддержку конфигурации-кандидата. Некоторые сетевые операционные системы, такие как Comware 7 компании HPE и IOS-XE компании Cisco, поддерживают протокол NETCONF, но в них пока еще не реализована поддержка хранилища данных конфигурации-кандидата.



Рекомендуется всегда выполнять тщательную проверку используемых аппаратных и программных платформ, даже если все они приобретены у одного поставщика/производителя. Всегда существует вероятность того, что предлагаемые ими функциональные возможности окажутся различными. Поддержка (или отсутствие поддержки) конфигурации-кандидата является всего лишь одним из примеров подобной ситуации.

Выше было отмечено, что при использовании конфигурации-кандидата вводятся различные конфигурации, но фактически они не применяются на устройстве до тех пор, пока не будет выполнена операция commit (прием-подтверждение). Следствием этого является наличие другой важнейшей функциональной характеристики NETCONF-совместимых устройств – внесения изменений в конфигурацию в форме транзакции (transaction). На практике это означает, что все объекты конфигурации (команды) принимаются и подтверждаются (committed) как транзакция. Таким образом, либо все команды успешно выполнены, либо они не применены вовсе. Это полностью противоречит более широко распространенной практике ввода последовательностей команд, когда в случае некорректности или сбоя одной из команд в середине последовательности конфигурация оказывается частично измененной, то есть, вероятнее, всего некорректной.

Поддержка конфигурации-кандидата – это всего лишь одна из функциональных характеристик NETCONF. В следующих разделах мы более глубоко рассмотрим стек протоколов NETCONF.

Функциональные особенности стека протоколов NETCONF

Выше были кратко описаны некоторые функциональные характеристики NETCONF, но теперь мы более подробно рассмотрим стек протоколов NETCONF, используемый для обмена данными между клиентом и сервером. В последующих примерах клиентом будет приложение на языке Python или SSH-клиент, а сервером – целевое сетевое устройство, которое необходимо автоматизировать.

В стеке протоколов NETCONF можно выделить четыре основных уровня (см. табл. 7.3). Мы рассмотрим каждый уровень и приведем примеры, показывающие смысловое значение уровней с представлением в форме XML-объекта содержимого, передаваемого между клиентом и сервером.

Tlgm: @it_boooks

Таблица 7.3. Уровни стека протоколов NETCONF

Уровень	Примеры
Транспортный	SSHv2, SOAP, TLS
Сообщения	`<rpc>`, `<rpc-reply>`
Операции	`<get-config>`, `<get>`, `<copy-config>`, `<lock>`, `<unlock>`, `<edit-config>`, `<delete-config>`, `<kill-session>`, `<close-session>`
Содержимое	Конфигурация/управление файлами: XML-представление моделей данных (YANG, XSD)



Для кодирования данных NETCONF поддерживает только формат XML. Но следует помнить, что RESTful API обеспечивают возможность поддержки как формата JSON, так и формата XML.

Транспортный уровень Чаще всего при реализации стека протоколов NETCONF используется SSH как транспортный протокол, при этом формируется собственная SSH-подсистема. Во всех приводимых ниже примерах используется NETCONF поверх протокола SSH, но технически вполне возможна реализация NETCONF поверх SOAP, TLS и любых других протоколов, соответствующих требованиям NETCONF. Поскольку протокол SOAP становится основой RESTful API, возникают некоторые ограничения в дальнейшей разработке NETCONF поверх SOAP. И, несмотря на техническую возможность реализации NETCONF поверх TLS, платформы, в настоящее время поддерживающие эту реализацию, в нашей книге не рассматриваются.

Ниже приведены некоторые требования NETCONF к транспортному протоколу:

- обязательная поддержка сеансов, ориентированных на соединение, следовательно, обеспечение постоянного соединения между клиентом и сервером;
- сеансы NETCONF обязательно должны обеспечивать средства аутентификации, сохранения целостности данных, секретности и защиты от повторной передачи перехваченных сообщений;
- несмотря на то что NETCONF может быть реализован на основе других транспортных протоколов, каждая реализация обязательно должна поддерживать, как минимум, протокол SSH.

Уровень сообщений Сообщения (messages) NETCONF основаны на механизме удаленного вызова процедур (remote procedure call – RPC) и на соответствующей модели обмена информацией с кодировкой каждого сообщения в формате XML. Модель, основанная на RPC, позволяет использовать XML-сообщения независимо от типа транспортного протокола. NETCONF поддерживает два типа сообщений: `<rpc>` и `<rpc-replay>`. Подробное изучение реального объекта в кодировке XML помогает лучше понять функционирование NETCONF, поэтому рассмотрим RPC-запрос NETCONF.

Проще говоря, любое сообщение всегда принадлежит к типу `<rpc>` или к типу `<rpc-replay>`. Этот факт обязательно обозначается XML-тегом самого верхнего уровня в кодируемом объекте.

Tlgm: @it_boooks


```
<rpc message-id="101">
     <!-- rest of request as XML... -->
 </rpc>
```

В NETCONF каждое сообщение <rpc> включает обязательный атрибут message-id, как показано в предыдущем примере. Это произвольная строка, которую клиент отправляет на сервер. Сервер использует этот идентификатор в заголовке ответа, чтобы клиент точно знал, на какое сообщение отвечает сервер.

Вторым типом сообщения является <rpc-replay>. NETCONF-сервер формирует ответ с использованием атрибута message-id и некоторыми другими атрибутами, полученными от клиента (например, пространства имен XML).

```
<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <data>
     <!-- XML content/response... -->
   </data>
 </rpc-reply>
```

В приведенном примере сообщения <rpc-replay> предполагается, что указанное пространство имен XML было передано клиентом в сообщении <rpc>. Отметим, что действительные данные ответа NETCONF-сервера размещаются внутри тега <data>.

Далее мы рассмотрим, каким образом запрос NETCONF определяет конкретную операцию (RPC), которая должна быть выполнена на сервере.

Уровень операций XML-элемент самого верхнего уровня всегда определяет тип передаваемого сообщения (<rpc> или <rpc-replay>). При отправке NETCONF-запроса от клиента серверу следующим элементом, то есть потомком типа сообщения, является затребованная операция NETCONF (RPC). Список операций NETCONF приведен в табл. 7.3, и мы рассмотрим каждую из этих операций более подробно.

Две основные операции, рассматриваемые в текущей главе: <get> и <edit-config>.

Операция <get> позволяет получить текущую рабочую конфигурацию и информацию о состоянии устройства.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <!-- XML content/response... -->
  </get>
</rpc>
```

Поскольку <get> является элементом-потомком в составе сообщения <rpc>, это означает, что клиент запрашивает выполнение операции <get> на NETCONF-сервере.

Внутри иерархии тега <get> размещается необязательный фильтр типов, позволяющий выборочно извлекать отдельные части текущей рабочей конфигурации. Это могут быть фильтры типа *subtree* или *xpath*. Мы подробно рассмотрим фильтры типа *subtree*, позволяющие получить XML-документ, яв-

Tlgm: @it_boooks

ляющийся поддеревом полного дерева XML-иерархии, в соответствии с условиями, заданными в запросе.

В следующем примере показана ссылка на конкретный объект данных в формате XML при помощи элемента <native> и URL http://cisco.com/ns/yang/ned/ios. Этот объект данных является XML-представлением определенной модели данных, существующей на целевом устройстве. Такая модель данных представляет полную рабочую конфигурацию в формате XML, но в примере запрашивается только часть общей иерархии, обозначенная тегом <interface>.



Во всех примерах текущей главы можно видеть, что реальные объекты в формате JSON или XML, передаваемые между клиентами и серверами, в высокой степени зависят от конкретного производителя оборудования и от используемой операционной системы.

В следующих двух примерах показана операция <get>, содержащая XML-запросы для устройства Cisco IOS-XE с использованием кода 16.3+.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <native xmlns="http://cisco.com/ns/yang/ned/ios">
       <interface>
       </interface>
      </native>
    </filter>
  </get>
</rpc>
```

В фильтры для XML-дерева можно добавить больше элементов, чтобы уменьшить объем данных ответа, возвращаемого NETCONF-сервером. Добавим в фильтр два элемента, чтобы вместо объектов конфигурации всех интерфейсов получить только конфигурацию для GigabitEthernet1.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
      <filter type="subtree">
        <native xmlns="http://cisco.com/ns/yang/ned/ios">
         <interface>
          <GigabitEthernet>
           <name>1</name>
          </GigabitEthernet>
         </interface>
        </native>
      </filter>
  </get>
</rpc>
```

Следующей наиболее часто применяемой операцией NETCONF является <edit-config>. Эта операция используется для внесения изменений в конфигурацию. Если говорить более точно, операция <edit-config> загружает конфигурацию в заданное хранилище данных конфигурации: текущая рабочая, начальная или конфигурация-кандидат.

Tlgm: @it_boooks

При использовании операции <edit-config> устанавливается целевое хранилище данных конфигурации с помощью тега <target>. Если тег не определен, то по умолчанию принимается хранилище данных текущей рабочей конфигурации. Кроме того, в иерархии тега <edit-config> содержатся элементы конфигурации, которые должны загружаться в указанное целевое хранилище данных. Эти элементы чаще всего включаются в элемент <config>. XML-элементы, содержащиеся в элементе <config>, отображаются обратно в элементы существующей модели данных.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <configuration>
        <routing-options>
          <static>
            <route>
              <name>0.0.0.0/0</name>
              <next-hop>10.1.0.1</next-hop>
            </route>
          </static>
        </routing-options>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

Элемент <config> не является обязательным компонентом операции <edit-config>. Возможное содержимое элемента <config> зависит от функциональных характеристик NETCONF, поддерживаемых конкретным сетевым устройством. Если поддерживается функциональная характеристика :url, то можно воспользоваться тегом <url> для определения места расположения файла, содержащего данные конфигурации.



Более подробно функциональные характеристики NETCONF рассматриваются немного позже в этом разделе.

Кроме того, производители оборудования могут реализовать дополнительные возможности, зависимые от платформы. Примером могут служить функциональные возможности, предлагаемые Juniper Junos для операции <edit-config>. В предыдущем примере использовался элемент <config> и явно указанные XML-объекты конфигурации для добавления статического маршрута в устройство Juniper Junos. Но устройство Juniper Junos также поддерживает в операции <edit-config> тег <config-text>, позволяющий включать элементы конфигурации в текстовом формате (в фигурных скобках или с помощью специально определенного синтаксиса).

Tlgm: @it_boooks

Операция <edit-config> также поддерживает атрибут operation, обеспечивающий бóльшую гибкость в применении объекта конфигурации на устройстве. При использовании атрибута operation можно установить для него одно из пяти значений: merge, replace, create, delete или remove. По умолчанию устанавливается значение merge. Если необходимо удалить маршрут из предыдущего примера, то можно воспользоваться значениями delete или remove. Различие между этими значениями заключается в том, что при заданном значении delete для несуществующего объекта генерируется ошибка. Можно использовать необязательное значение create, но если заданный объект уже существует, то генерируется ошибка. Поэтому чаще применяется значение merge для внесения изменений в конфигурацию.

Наконец, можно применить значение replace, если необходимо заменить XML-иерархию в объекте данных конфигурации. В примере с определением статического маршрута можно воспользоваться значением replace, если необходимо в конечном итоге ограничиться только статическим маршрутом по умолчанию, определенным на устройстве. При этом будут автоматически удалены все другие сконфигурированные статические маршруты.



Если вам пока еще не все понятно, не стоит беспокоиться. Когда мы приступим к изучению автоматизации устройств с использованием NETCONF в следующих двух разделах, вам будет предложено множество практических примеров, в которых разнообразные XML-объекты применяются к различным типам устройств, а также используются операции NETCONF, такие как merge и replace.

Ранее уже было показано, как выглядят XML-документы при использовании операций <get> и <edit-config>. В следующем списке кратко описаны основные операции NETCONF:

- <get-config> – получение всех частей заданной конфигурации (рабочей, начальной или конфигурации-кандидата);
- <copy-config> – создание или замена хранилища данных конфигурации с содержимым, взятым из другого хранилища данных конфигурации. Эта операция требует использования полной конфигурации;
- <delete-config> – удаление хранилища данных конфигурации (хранилище текущей рабочей конфигурации не может быть удалено);
- <lock> – блокировка системы хранилища данных конфигурации для обновляемого устройства для гарантии того, что никакие другие системы (клиенты NETCONF) не смогут вносить изменения во время обновления;
- <unlock> – снятие ранее установленной блокировки на хранилище данных конфигурации;
- <close-session> – запрос на аккуратное и корректное завершение сеанса NETCONF;
- <kill-session> – немедленное принудительное завершение сеанса NETCONF.

Разумеется, этот список неполон, но в нем приведены все ключевые операции, которые обязательно должно поддерживать каждое сетевое устройство

Tlgm: @it_boooks

при реализации NETCONF. Серверы NETCONF также могут поддерживать дополнительные операции, такие как <commit> и <validate>. Для обеспечения функциональности подобных дополнительных операций устройство непременно должно поддерживать требуемые зависимости, использующие внутренние функции NETCONF.

Операция <commit> принимает и подтверждает конфигурацию-кандидат как новую рабочую конфигурацию устройства. Для обеспечения функциональности операции <commit> устройство обязательно должно поддерживать функциональную возможность :candidate.

Операция <validate> проверяет корректность содержимого заданной конфигурации (рабочей, кандидата, начальной). Сама процедура проверки (валидации) состоит из операций проверки конфигурации на синтаксис и на семантику перед применением этой конфигурации на устройстве.



Функциональные характеристики (возможности) NETCONF были упомянуты уже дважды, поэтому требуется кратко пояснить, что они собой представляют. Как вы уже знаете, NETCONF поддерживает основной набор RPC-операций. Но эти операции реализованы через поддерживаемый устройством базовый набор функциональных характеристик (возможностей) (capabilities) NETCONF. Этими функциональными характеристиками (возможностями) обмениваются клиент и сервер во время установления соединения, а поддерживаемые функциональные характеристики обозначаются с помощью URL/URI. Например, каждое устройство, совместимое с NETCONF, должно поддерживать базовую операцию, обозначенную с помощью URL `urn:ietf:params:xml:ns:netconf:base:1.0`. Дополнительные функциональные характеристики используют форму `urn:ietf:params:netconf:capability:{`*name*`}:1.x`, где *name* – это имя функциональной характеристики, а конкретные функциональные характеристики обычно обозначаются как `:name`, и выше это было показано на примере обращения к характеристике `:candidate`. Когда мы начнем изучение NETCONF с практической точки зрения, вы увидите в действии все функциональные характеристики (возможности), поддерживаемые конкретными устройствами.

Уровень содержимого (контента) Последним из рассматриваемых в текущем разделе уровней стека протоколов NETCONF является уровень содержимого или контента (content). Содержимое представлено реальным XML-документом, который размещается в тегах элементов RPC-операции. Выше были показаны примеры включения контента в конкретные операции NETCONF.

В первом примере рассматривается контент, в котором избирательно запрашиваются элементы конфигурации для интерфейсов на устройстве Cisco IOS-XE:

```
<native xmlns="http://cisco.com/ns/yang/ned/ios">
  <interface>
  </interface>
</native>
```

Здесь самое важное – понять, что контент является XML-представлением конкретной схемы или модели данных, поддерживаемой запрашиваемым устройством. Схемы и модели данных рассматривались в главе 5.

Tlgm: @it_boooks

Используемое в примерах устройство IOS-XE поддерживает модели, написанные на языке YANG, а одна из моделей является представлением полной текущей рабочей конфигурации. Несмотря на то что модель написана на языке YANG, она непременно должна быть представлена в формате XML при обмене данными между клиентом и сервером, поскольку NETCONF поддерживает только формат кодирования XML.

В следующем примере показан контент для добавления статического маршрута в устройство Juniper Junos. И здесь чрезвычайно важно понять, как сформировать корректный XML-документ, соответствующий требованиям ОС устройства, при этом знание языка, такого как YANG или XML Schema Definitions (XSD), на котором написана модель или схема, гораздо менее важно. Например, если имеется приведенный в следующем примере Juniper XML-документ, то можно ли узнать, отобразится он в схему XSD или в модель, написанную на языке YANG? Право ответа на этот вопрос мы предоставляем читателю в качестве дополнительного упражнения.

```
<configuration>
  <routing-options>
    <static>
      <route>
        <name>0.0.0.0/0</name>
        <next-hop>10.1.0.1</next-hop>
      </route>
    </static>
  </routing-options>
</configuration>
```

После знакомства с API, рассматриваемыми в текущей главе, мы переходим к изучению их практического использования.

Практическое использование сетевых API

Поскольку мы начинаем изучение с общих принципов применения и взаимодействия с сетевыми API, главное внимание в этом разделе, как и на протяжении всей книги, уделено независимым от конкретных производителей оборудования инструментальным средствам и библиотекам. В частности, мы будем рассматривать такие инструменты, как cURL и Postman, для работы с API на основе протокола HTTP, а также методику «NETCONF поверх SSH» для работы с NETCONF API на сетевых устройствах.

Важно отметить, что в текущем разделе рассматривается только «потребительское» использование сетевых API, то есть демонстрируются способы, позволяющие приступить к применению и тестированию сетевых API без написания какого-либо кода. Главная цель этого раздела – начать практически применять все полученные до этого момента знания о конкретных типах API. В разделе не рассматриваются инструменты и методики, которые можно было бы применять для автоматизации реально эксплуатируемых сетей. Такие типы

Tlgm: @it_boooks

инструментальных средств и библиотек будут рассматриваться в следующем разделе «Автоматизация с использованием сетевых API».

Практическое использование API на основе протокола HTTP

Начнем с изучения практического применения API на основе протокола HTTP. Сразу отметим, что и для RESTful API, и для API, несовместимых с REST, используются одни и те же инструментальные средства. Первый рассматриваемый инструмент называется cURL.

cURL

cURL – это инструмент (утилита) командной строки для работы с URL. Это означает, что, применяя программу cURL в командной строке ОС Linux, можно отправлять запросы протокола HTTP. Несмотря на то что утилита cURL работает с URL (как понятно по ее названию), ее можно использовать для обмена данными с серверами, работающими не только по протоколу HTTP, но и поддерживающими такие протоколы, как FTP, SFTP, TFTP, TELNET, и многие другие. Воспользуйтесь командой `man curl` в командной строке Linux, чтобы получить подробнейшее описание всех функций и возможностей, предоставляемых программой cURL.

Мы будем рассматривать взаимодействие cURL с программным интерфейсом Cisco ASA RESTful API. Так как мы только начинаем изучение RESTful API, в первом примере выполним простой запрос HTTP GET, позволяющий получить данные обо всех интерфейсах устройства защиты Cisco ASA.



Платформы Cisco ASA и ASAv, поддерживающие RESTful API, предлагают встроенную документацию и консоль для тестирования API непосредственно на ASA-устройстве. Доступ к документации осуществляется по адресу https://<asa-ip>/doc, необходимо зарегистрироваться, войти, затем просматривать документацию и тестировать любые API, которые поддерживает данное ASA-устройство.

Для выполнения обращения (вызова) к API с целью получения списка интерфейсов и их конфигураций воспользуемся утилитой cURL, как показано в следующем примере:

```
$ curl -u ntc:ntc123 -k https://asav/api/interfaces/physical
```

В команде используются два флага, а также указывается требуемый URL, необходимый для получения данных об интерфейсах на ASA-устройстве. Первый флаг `-u` сообщает, что за ним следует комбинация *`user_name:_password_`*. Второй флаг `-k` в явной форме позволяет программе cURL устанавливать незащищенные (insecure) SSL-соединения, а в нашем случае необходимо разрешение на использование самоподписанного сертификата на ASA-устройстве. В конце команды указан URL запрашиваемого ресурса.

```
$ curl -u ntc:ntc123 -k https://asav/api/interfaces/physical
# ответ пропущен
```

Tlgm: @it_boooks

Если вы выполните приведенную выше команду запуска cURL, то получите длинный многоэкранный вывод на экран терминала, который трудно читать. Но можно направить вывод этой команды через конвейер (pipe) на ввод другой команды `python -m json.tool`, чтобы получить отформатированный объект, содержащийся в ответе, который гораздо удобнее изучать.

```
$ curl -u ntc:ntc123 -k https://asav/api/interfaces/physical | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  4114  100  4114    0     0  15819      0 --:--:-- --:--:-- --:--:-- 15884
{
   "kind":"collection#MgmtInterface",
   "selfLink":"https://asav/api/interfaces/physical",
   "rangeInfo":{
      "offset":0,
      "limit":4,
      "total":4
   },
   "items":[
      {
         "kind":"object#MgmtInterface",
         "selfLink":"https://asav/api/interfaces/physical/Management0_API_SLASH_0",
         "hardwareID":"Management0/0",
         "interfaceDesc":"",
         "channelGroupID":"",
         "channelGroupMode":"active",
         "duplex":"auto",
         # keys removed for brevity
         "managementOnly":true,
         "mtu":1500,
         "name":"management",
         "securityLevel":0,
         "shutdown":false,
         "speed":"auto",
         "ipAddress":{
            "kind":"StaticIP",
            "ip":{
               "kind":"IPv4Address",
               "value":"10.0.0.101"
            },
            "netMask":{
               "kind":"IPv4NetMask",
               "value":"255.255.255.0"
            }
         },,
         "objectId":"Management0_API_SLASH_0"
      },
      {
         "kind":"object#GigabitInterface",
         "selfLink":"https://asav/api/interfaces/physical/GigabitEthernet0_API_SLASH_0",
         "hardwareID":"GigabitEthernet0/0",
```

Tlgm: @it_boooks


```
        "interfaceDesc":"",
        # ключи удалены в целях экономии места
      },
      # словари других интерфейсов удалены в целях экономии места
   ]
}
$
```

Отметим, что тип этого объекта также получен и выведен на экран терминала. Здесь можно видеть, что это объект в формате JSON (или словарь в терминологии языка Python), так как он начинается открывающей, а заканчивается закрывающей фигурной скобкой. Также можно заметить, что ключ `items` – это список словарей, в котором каждый словарь представляет отдельный интерфейс.

Cisco ASA API поддерживает только кодировку в формате JSON, но многие RESTful API поддерживают и формат JSON, и формат XML. Если системы или сетевые устройства поддерживают различные типы кодировок, то необходимо явно определять требуемый тип кодировки в тексте HTTP-запроса. Этот тип объявления включается в заголовок HTTP, который является частью HTTP-запроса.

Чаще всего применяются два заголовка HTTP, которые мы будем использовать при изучении API на основе HTTP: `Accept` и `Content-Type`. Заголовок `Accept` применяется для определения конкретных типов носителей, которые являются приемлемыми (разрешенными) для ответа на выполняемый запрос. Заголовок `Content-Type` используется для определения типа данных, отправляемых на сервер для внесения изменений в ресурс. В следующих двух подразделах будут рассмотрены примеры практического применения этих заголовков HTTP.

Выше был приведен простой пример выполнения запроса GET к ASA RESTful API с использованием программы cURL. Далее мы рассмотрим более дружественное к пользователю инструментальное средство Postman для исследования и тестирования API на основе протокола HTTP.

Взаимодействие с API на основе протокола HTTP с помощью программы Postman

Postman – это приложение браузера Google Chrome, предоставляющее удобный и понятный графический пользовательский веб-интерфейс для взаимодействия с API на основе протокола HTTP. Как вы вскоре увидите, Postman и прочие подобные инструменты позволяют существенно упростить изучение и тестирование HTTP API, поскольку основное внимание сосредоточено на обеспечении возможности использования API без необходимости написания кода.

После установки Postman и запуска этой программы как приложения браузера Chrome пользователю предлагается экран интерфейса, похожий на рис. 7.2.

Tlgm: @it_boooks

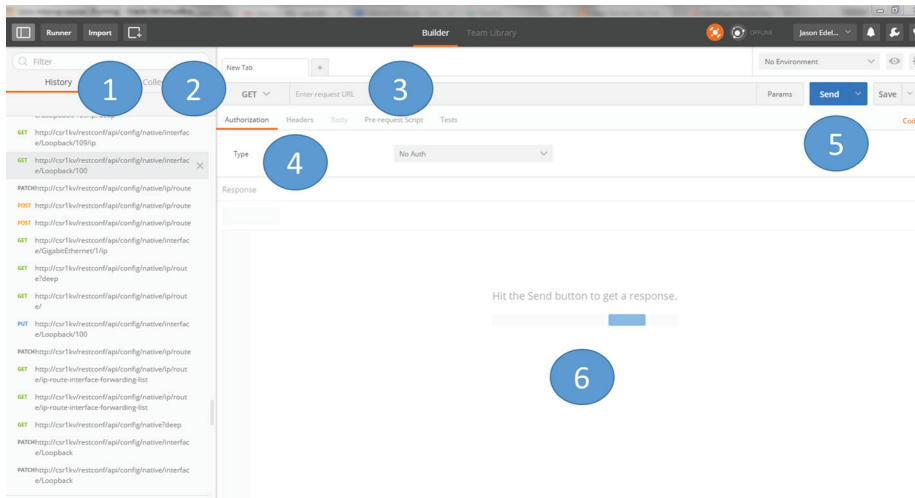


Рис. 7.2 ❖ Общий интерфейс приложения Postman

Рассмотрим более подробно элементы интерфейса, используя числовые указатели, размещенные на рис. 7.2:

1. В левой панели размещены две вкладки: **History** и **Collections**. На вкладке **History** (Хронология) показаны все ранее выполненные вызовы API с использованием программы Postman. Кроме того, к хронологии можно легко получить доступ, если вы зарегистрировались в Postman на другом компьютере, при этом браузер Chrome выполняет эту функцию точно так же, как доступ к хронологии веб-браузера. Вкладка **Collections** (Наборы) позволяет сохранять последовательности запросов к API как набор (collection) и после создания выполнять эти запросы отдельно или весь набор в целом, как аналог скрипта.
2. В этом спускающемся меню можно выбрать требуемый метод запроса HTTP (например, GET, PUT, PATCH, POST, DELETE). После входа в программу Postman по умолчанию выбран метод GET.
3. Здесь вводится URL, необходимый для выполнения требуемого запроса к API.
4. На этой иллюстрации мы сосредоточимся на трех вкладках в этой области: **Authorization**, **Headers** и **Body**. На вкладке **Authorization** (Авторизация) выбирается соответствующая форма аутентификации, требуемая сервером. В наших последующих примерах для работы с Cisco ASA RESTful API используется вариант Basic Auth. На вкладке **Headers** (Заголовки) определяются заголовки запросов, применяемые в конкретном запросе к API. Например, если нужно определить тип кодировки запроса, требуемый вариант выбирается именно в этой вкладке. На вкладке **Body** (Тело) определяется тип кодировки тела сообщения – JSON или XML, – которое отправляется в запросе к API, если вносятся изменения в ресурс, напри-

Tlgm: @it_boooks

мер использование метода POST для создания виртуального интерфейса типа «обратная петля» (loopback).

5. После определения требуемого метода HTTP (2), ввода целевого URL (3) и конфигурирования необходимых параметров (4) можно щелкнуть по кнопке **Send** (Отправить) для выполнения вызова API на указанном устройстве.
6. После завершения выполнения запроса API результат выводится в этой текстовой панели.

В первом примере использования приложения Postman демонстрируется, как получить список интерфейсов на устройстве Cisco ASA с помощью поддерживаемого RESTful API. Это тот же самый простой пример, который был выполнен с помощью программы cURL в предыдущем разделе.

На рис. 7.3 показаны установка метода HTTP GET, ввод целевого URL и прочих требуемых параметров.

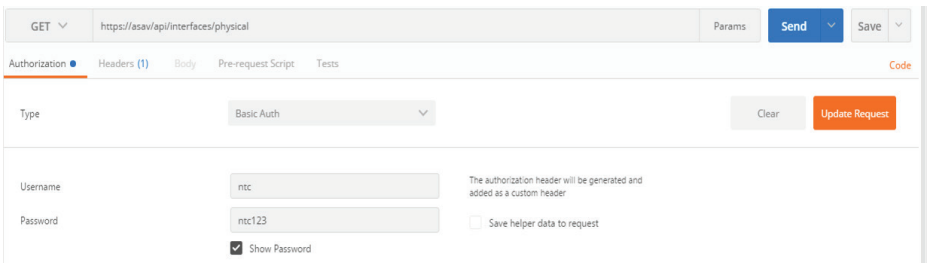


Рис. 7.3 ❖ Выполнение запроса GET в приложении Postman

Отметим, что вкладка **Headers** теперь имеет вид **Headers (1)**, как показано на рис. 7.4. Причина в том, что была нажата кнопка **Update Request** (Запрос на обновление). При этом автоматически создается заголовок с именем Authorization, в котором содержится закодированная в формате base64 строка, состоящая из имени пользователя и пароля.

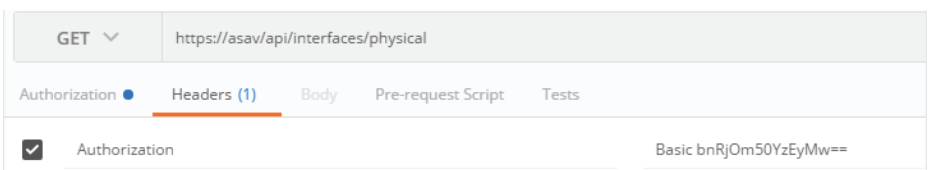


Рис. 7.4 ❖ Заголовок Authorization для выполняемого запроса

Строка, закодированная в формате base64, не является зашифрованной. Строки в кодировке base64 с легкостью кодируются и декодируются с помощью модуля языка Python `base64`.

Tlgm: @it_boooks


```
>>> import base64
>>>
>>> encoded = base64.b64encode('ntc:ntc123')
>>> encoded
'bnRjOm50YzEyMw=='
>>>
>>> text = base64.b64decode(encoded)
>>> text
'ntc:ntc123'
>>>
```

После того как запрос сформирован, он отправляется на устройство щелчком по синей кнопке **Send** (Отправить). Когда сервер выполнит запрос, в нижней части экрана выводится его ответ и код состояния HTTP, как показано на рис. 7.5.

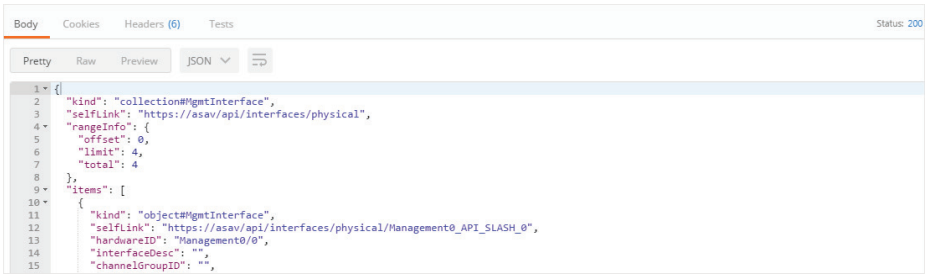


Рис. 7.5 ❖ Просмотр ответа на запрос HTTP GET

Устройство Cisco ASA поддерживает только формат JSON, тем не менее заголовок `Accept` запроса HTTP может быть явно определен на вкладке **Headers**, хотя это не повлияет на конечный результат (см. рис. 7.6).

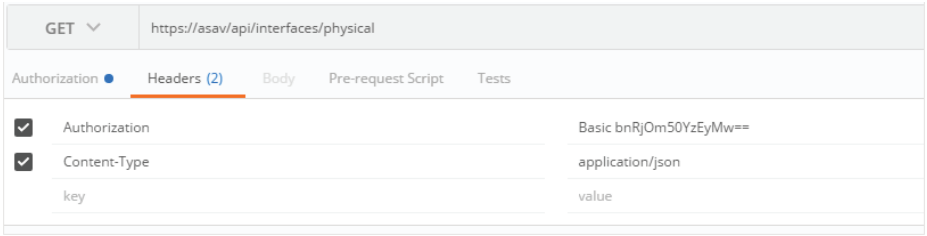


Рис. 7.6 ❖ Определение заголовка запроса

Процедура получения конфигурационных данных с помощью запроса GET вполне понятна, так как для нее требуются только регистрационные данные (имя и пароль) и URL. При внесении изменений в конфигурацию потребуется HTTP Body (Тело) для обращения к соответствующему методу API.

Tlgm: @it_boooks

После того как в предыдущем примере был создан простой запрос, извлекающий имена всех физических интерфейсов, можно перейти к внесению изменений в один из этих интерфейсов. Сконфигурируем описание интерфейса для GigabitEthernet0/0. Чтобы выполнить такую корректировку, потребуется поменять три элемента вызова API в приложении Postman:

1. Изменить URL.
2. Изменить метод запроса HTTP.
3. Добавить соответствующее тело запроса в формате JSON.

Возможно еще и четвертое необязательное изменение – настройка заголовка *Content-Type* (*Тип содержимого*), но поскольку ASA поддерживает только формат JSON, этого можно не делать.

Новый вызов API можно определить следующим образом:

1. Тип запроса HTTP: PATCH.
2. URL: https://asav/api/interfaces/physical/GigabitEthernet0_API_SLASH_0.
3. Тело:

```
{
  "kind": "object#GigabitInterface",
  "interfaceDesc": "Configured by Postman"
}
```

После внесения вышеперечисленных изменений в приложении Postman вид экрана будет таким, как показано на рис. 7.7.

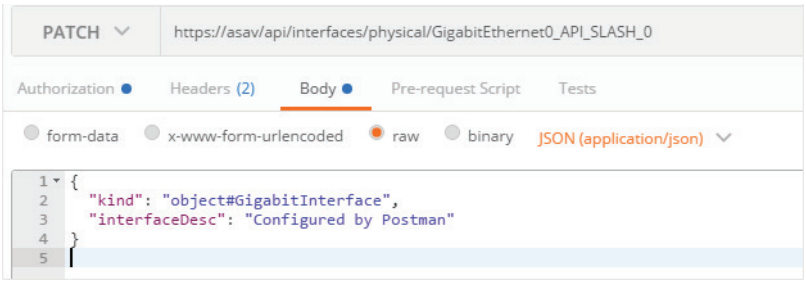


Рис. 7.7 ❖ Выполнение запроса PATCH в приложении Postman

После щелчка по кнопке **Send** и выполнения запроса на устройстве Cisco ASA появляется заново сконфигурированное описание интерфейса.

Научиться правильному формированию запросов API можно, лишь внимательно изучая документацию на используемый API. В документации (определение API и подробные характеристики вызываемых методов и функций) определено, каким должен быть целевой URL, тип запроса HTTP, заголовки, а также какое тело (содержимое) необходимо для успешного вызова API.

Например, каким образом нам стало известно, что при внесении изменений в интерфейс устройства ASA нужно использовать URL GigabitEthernet0_API_SLASH_0, что требуется запрос PATCH и необходимо передать ключи `kind` и `interfaceDesc` в теле запроса в фор-

Tlgm: @it_boooks

мате JSON? Ответ прост: документация API. К счастью, документация для ASA, как уже было сказано ранее, размещена непосредственно на устройстве по адресу https://<asa-ip-address>/doc/. Кроме того, можно также получить некоторую информацию из результатов выполнения запросов GET, поскольку чаще всего требуется использование одних и тех же значений в теле запросов POST/PATCH/PUT, как это было видно на примере использования значений ключей `object#GigabitInterface` и `kind`.

Итак, мы рассмотрели два инструментальных средства, позволяющих взаимодействовать с API на основе протокола HTTP без написания какого-либо программного кода. Были продемонстрированы способы выполнения одного и того же вызова API с использованием утилиты cURL и приложения Postman, а также внесение изменений в конфигурацию с помощью Poistman. Теперь мы переходим к изучению практического использования NETCONF.

Практическое использование NETCONF

При изучении нового API предпочтительнее начать с инструментальных средств, совместимых с этим API, которые позволяют осваивать его без необходимости написания программного кода. Это можно было наблюдать на примере приложения Postman при изучении API, основанных на протоколе HTTP. Для NETCONF мы рассмотрим использование SSH-клиента, который создает интерактивный сеанс NETCONF. Вы узнаете, как сформировать правильный запрос NETCONF, а также увидите, как устройство отвечает на принятый запрос. При этом не потребуется ни одной строчки программного кода.



Использование интерактивного режима NETCONF в сеансе SSH не представляет собой идеальный вариант для изучения NETCONF. Такой подход не является ни интуитивно понятным, ни дружественным к пользователю, но других инструментальных средств для удобного изучения и освоения практического применения NETCONF просто не существует. В любом случае, не следует рассматривать интерактивный сеанс SSH как единственную правильную эксплуатационную модель для автоматизации сетевых устройств с использованием NETCONF.

В первом примере устанавливается соединение с маршрутизатором Cisco IOS-XE по протоколу SSH с использованием порта 830, который является номером порта по умолчанию для NETCONF. Для этого применяется стандартная команда ОС Linux `ssh`, но в ней явно указывается номер порта 830.

```
$ ssh -p 830 ntc@ios-csr1kv
```

Сразу после установления соединения и выполнения процедуры аутентификации сервер NETCONF (маршрутизатор Cisco IOS-XE) отвечает сообщением типа hello, которое включает все поддерживаемые функциональные возможности NETCONF, в том числе поддерживаемые операции, характеристики и свойства, модели/схемы, а также идентификатор сеанса (session ID).

Ниже приведен фрагмент ответа, в котором перечислены функциональные возможности устройства Cisco IOS-XE. Выводимая информация немного под-

Tlgm: @it_boooks

редактирована, так как в действительности ответ содержит несколько сотен строк из-за большого количества поддерживаемых моделей:

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:base:1.1</capability>
<capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
<capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.1</capability>
<capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
<capability>urn:ietf:params:netconf:capability:notification:1.0</capability>
<capability>urn:ietf:params:netconf:capability:interleave:1.0</capability>
<capability>http://tail-f.com/ns/netconf/actions/1.0</capability>
<capability>http://tail-f.com/ns/netconf/extensions</capability>
<capability>urn:ietf:params:netconf:capability:with-defaults:1.0?basic-mode=
report-all</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-with-defaults?revision=
2011-06-01&amp;module=ietf-netconf-with-defaults</capability>
<capability>http://cisco.com/ns/example/enable?module=enable</capability>
<capability>http://cisco.com/ns/yang/ned/ios?module=ned&amp;revision=2016-07-01
</capability>
<capability>http://cisco.com/ns/yang/ned/ios/asr1k?module=ned-asr1k&amp;revision=
2016-04-07</capability>
<capability>http://cisco.com/yang/cisco-ia?module=cisco-ia&amp;revision=
2016-05-20</capability>
<!-- ЧАСТЬ ВЫВОДИМЫХ ДАННЫХ ПРОПУЩЕНА -->
<capability>urn:ietf:params:xml:ns:yang:smiv2:VPN-TC-STD-MIB?module=
VPN-TC-STD-MIB&amp;revision=2005-11-15</capability>
</capabilities>
<session-id>324</session-id></hello>]]>]]>
```

После приема списка функциональных возможностей сервера начинается процесс настройки соединения с NETCONF. Следующий шаг – отправка информации о функциональных возможностях клиента (то есть поддерживаемых на нашей стороне). Обмен списками поддерживаемых функциональных возможностей необходим для обеспечения возможности отправки правильных NETCONF-запросов на сервер.

В этом примере функциональные возможности клиента ограничены несколькими основными операциями и поддержкой только одной модели на устройстве IOS XE.

Объект `hello`, который необходимо отправить на устройство для завершения процедуры обмена списками поддерживаемых функциональных возможностей, выглядит следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
```

Tlgm: @it_boooks


```
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.1</capability>
    <capability>http://cisco.com/ns/yang/ned/ios?module=ned&amp;revision=2016-07-01</
capability>
    <capability>http://cisco.com/ns/yang/ned/ios/asr1k?module=ned-
asr1k&amp;revision=2016-04-07</capability>
  </capabilities>
</hello>
]]>]]>
```



Когда клиент отправляет свое сообщение hello, в нем не передается атрибут «идентификатор сеанса» (session ID), который был передан в сообщении hello от сервера.



Обратите внимание на последние шесть символов в приведенном выше XML-документе:]]>]]>. Эти символы применяются для обозначения завершения запроса и сообщают о том, что можно начать обработку данного запроса. При работе в интерактивном сеансе NETCONF символы завершения запроса всегда являются обязательными.

Как только вы начнете работу с SSH-клиентом, сразу обнаружите, что он не похож на привычный интерактивный интерфейс командной строки (CLI), хотя представляет собой интерактивный сеанс. Здесь нет никаких средств подсказки и помощи, недоступен вызов экрана справки при нажатии клавиши со знаком вопроса. Недоступны также и страницы руководства `man`. Часто может складываться впечатление, что сеанс прерван или терминал завис. На самом деле это не так. Если после копирования и вставки XML-документов в терминале сеанса не выводятся сообщения об ошибках, то сеанс, вероятнее всего, идет в нормальном режиме. Чтобы прервать интерактивный сеанс, необходимо воспользоваться комбинацией клавиш **Ctrl**+**C** на клавиатуре, так как другого способа корректного завершения интерактивного NETCONF-сеанса нет.

После отправки клиента списка своих функциональных возможностей на сервер можно начинать передавать запросы NETCONF. Для предварительного формирования соответствующих XML-документов можно воспользоваться любым текстовым редактором.

Если вы практически выполняете предложенный здесь пример, то после отправки клиентского объекта `hello` и вставки его в терминале сеанса SSH вы должны увидеть следующую выведенную информацию (концовку сообщения `hello` от сервера и полное сообщение `hello` клиента, завершающееся символами]]>]]>):

```
<--- output truncated --->
<capability>urn:ietf:params:xml:ns:yang:smiv2:UDP-MIB?module=UDP-
MIB&amp;revision=2005-05-20</capability>
<capability>urn:ietf:params:xml:ns:yang:smiv2:VPN-TC-STD-MIB?module=VPN-TC-STD-
MIB&amp;revision=2005-11-15</capability>
</capabilities>
<session-id>1415</session-id></hello>]]>]]><?xml version="1.0"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <capabilities>
```

Tlgm: @it_boooks


```
        <capability>urn:ietf:params:netconf:base:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:writable-running:1.0
        </capability>
        <capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:validate:1.0
        </capability>
        <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0
        </capability>
        <capability>http://cisco.com/ns/yang/ned/ios?module=ned&amp;revision=
        2016-06-20</capability>
    </capabilities>
</hello>]]>]]>
```

Соединение с устройством успешно установлено, произведен обмен списками функциональных возможностей. Теперь можно выполнить требуемый запрос NETCONF. В первом примере будет показан запрос к устройству GigabitEthernet1 на получение его конфигурации.

Приведенный ниже XML-документ создан в текстовом редакторе, затем скопирован и вставлен в терминал интерактивного сеанса:

```
<?xml version="1.0"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
      <filter type="subtree">
        <native xmlns="http://cisco.com/ns/yang/ned/ios">
         <interface>
          <GigabitEthernet>
           <name>1</name>
          </GigabitEthernet>
         </interface>
        </native>
      </filter>
  </get>
</rpc>
]]>]]>
```

После нажатия клавиши **Enter** запрос передается на устройство. Пользователь получает ответ XML RPC практически в реальном времени.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
<data><native xmlns="http://cisco.com/ns/yang/ned/ios"><interface>
<GigabitEthernet><name>1</name><negotiation><auto>true</auto></negotiation>
<vrf><forwarding>MANAGEMENT</forwarding></vrf><ip><address><primary>
<address>10.0.0.51</address><mask>255.255.255.0</mask></primary></address></ip>
</GigabitEthernet></interface></native></data></rpc-reply>]]>]]>
```

Можно дополнительно воспользоваться функцией `<rpc-replay>` и любым инструментом форматирования XML, чтобы представить полученный ответ в более удобном для чтения виде. Псоле форматирования принятые данные выглядят следующим образом:

Tlgm: @it_boooks


```
<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
        <native xmlns="http://cisco.com/ns/yang/ned/ios">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                    <negotiation>
                        <auto>true</auto>
                    </negotiation>
                    <vrf>
                        <forwarding>MANAGEMENT</forwarding>
                    </vrf>
                    <ip>
                        <address>
                            <primary>
                                <address>10.0.0.51</address>
                                <mask>255.255.255.0</mask>
                            </primary>
                        </address>
                    </ip>
                </GigabitEthernet>
            </interface>
        </native>
    </data>
</rpc-reply>
```

Первый запрос NETCONF к сетевому устройству успешно выполнен, и получен корректный ответ. В этом примере не производились какие-либо действия ни с самим устройством, ни с полученными от него данными, как это делалось с помощью приложения Postman в предыдущем разделе. Здесь важно то, что протестирован и проверен XML-запрос, обеспечивающий получение конфигурации интерфейса GigabitEthernet1, и теперь мы точно знаем, как выглядит ответ, что позволяет упростить автоматизацию устройств с помощью языка Python.

Рассмотрим другой пример с применением интерактивного сеанса NETCONF поверх SSH. В этот раз мы будем работать с устройством Juniper vMX под управлением Junos и попробуем получить конфигурацию для интерфейса fxp0.

И в этом случае необходимо использовать процесс, описанный выше. Требуется установить соединение с подсистемой NETCONF по протоколу SSH, получить список функциональных возможностей сервера в сообщении `hello`, затем передать собственное сообщение `hello`, содержащее список функциональных возможностей клиента.

```
$ ssh -p 830 ntc@junos-vmx -s netconf
Password:
<!-- No zombies were killed during the creation of this user interface -->
<!-- user ntc, class j-super-user -->
<!-- Ни один зомби не был уничтожен при создании этого пользовательского интерфейса -->
<!-- пользователь ntc, класс j-super-user -->
```

Tlgm: @it_boooks


```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=
    http,ftp,file</capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-
    commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=
    http,ftp,file</capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>4128</session-id>
</hello>
]]>]]>
```



В зависимости от конкретной реализации производителя сетевого оборудования может потребоваться ключ `-s netconf` при установлении SSH-соединения с целевым устройством. Ключ `-s` определяет подсистему, используемую в сеансе SSH.

Затем клиент отвечает своим сообщением `hello`. В примере используются только основные функциональные возможности, так как не планируется делать ничего такого, что выходило бы за рамки поддерживаемых NETCONF основных операций и характеристик.

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
  </capabilities>
</hello>
]]>]]>
```

После завершения обмена списками функциональных возможностей на сервер передается XML-объект с соответствующим запросом конфигурации интерфейса fxp0.

```
<?xml version="1.0"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get>
        <filter type="subtree">
            <configuration>
                <interfaces>
```

Tlgm: @it_boooks


```
                    <interface>
                        <name>fxp0</name>
                    </interface>
                </interfaces>
            </configuration>
        </filter>
    </get>
</rpc>
]]>]]>
```

Ответ сервера приведен ниже:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1F4/junos" message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
junos:changed-seconds="1482848933" junos:changed-localtime="2016-12-27 14:28:53 UTC">
    <interfaces>
        <interface>
            <name>fxp0</name>
            <unit>
                <name>0</name>
                <family>
                    <inet>
                        <address>
                            <name>10.0.0.31/24</name>
                        </address>
                    </inet>
                </family>
            </unit>
        </interface>
    </interfaces>
</configuration>
<database-status-information>
<database-status>
<user>ntc</user>
<terminal>p0</terminal>
<pid>4091</pid>
<start-time junos:seconds="1482857982">2016-12-27 16:59:42 UTC</start-time>
<idle-time junos:seconds="52">00:00:52</idle-time>
<edit-path>[edit]</edit-path>
</database-status>
<database-status>
<user>ntc</user>
<terminal></terminal>
<pid>4168</pid>
<start-time junos:seconds="1482858044">2016-12-27 17:00:44 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
]]>]]>
```

Tlgm: @it_boooks

Мы рассмотрели два примера использования NETCONF-операций <get> на целевом устройстве. Теперь рассмотрим еще один пример, демонстрирующий применение операции <edit-config> для внесения изменений в конфигурацию.

Несмотря на то что сейчас все внимание сосредоточено на процедуре изменения конфигурации, мы, чтобы показать правильный способ формирования XML-запроса на внесение изменений, сначала выполним запрос get, позволяющий получить полную структуру объекта, который предполагается вернуть на устройство в измененном виде.



В этом примере применяется модель данных, соответствующая отраслевому стандарту для интерфейсов, под названием ietf-interfaces. Эта модель данных поддерживается Cisco IOS-XE.

После установления NETCONF-соединения с целевым устройством можно выполнить следующий запрос get, чтобы получить конфигурации интерфейсов.

```
<?xml version="1.0"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get>
        <filter type="subtree">
            <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
            </interfaces>
        </filter>
    </get>
</rpc>
]]>]]>
```

Слегка подредактированный и отформатированный ответ маршрутизатора Cisco IOS-XE приведен ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <name>GigabitEthernet1</name>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">
ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
          <address>
            <ip>10.0.0.51</ip>
            <netmask>255.255.255.0</netmask>
          </address>
        </ipv4>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
      </interface>
      <interface>
        <name>GigabitEthernet4</name>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">
```

Tlgm: @it_boooks


```
ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
      </interface>
    </interfaces>
  </data>
</rpc-reply>
]]>]]>
```



Приведенный в предыдущем примере ответ маршрутизатора сокращен – удалены данные об интерфейсах GigabitEthernet2 и GigabitEthernet3, чтобы сделать текст ответа более удобным для чтения.

Следует особо отметить, что для интерфейса GigabitEthernet1 существует возможность конфигурирования IP-адреса. Можно выполнить запрос `get`, проверить полученные в ответ объекты конфигурации и использовать данные ответа как основу для изменения данных конфигураций для других интерфейсов, чтобы упростить процесс в целом.

Рассмотрим процедуру внесения изменений в конфигурацию и конфигурирование IP-адреса 10.4.4.1/24 для интерфейса GigabitEthernet4.

Для формирования требуемого объекта необходимые данные будут извлечены из результата ранее выполненного запроса `get`. Изменения вносятся в следующие два элемента:

1. Объект, возвращенный в теге `<data>`, включается в тег `<config>`, если необходимо изменить данные конфигурации с использованием NETCONF-операции `<edit-config>`.
2. В формируемом объекте необходимо определить целевое (target) хранилище данных конфигурации (то есть текущей рабочей, начальной или конфигурации-кандидата) на основе информации о поддержке хранилищ данных целевым узлом.

После внесения этих изменений в конфигурацию результат выглядит следующим образом:

```
<?xml version="1.0"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <edit-config>
      <target>
        <running/>
      </target>
        <config>
          <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
            <interface>
              <name>GigabitEthernet4</name>
              <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">
ianaift:ethernetCsmacd</type>
              <enabled>true</enabled>
              <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
                <address>
```

Tlgm: @it_boooks


```
                <ip>10.4.4.1</ip>
                <netmask>255.255.255.0</netmask>
              </address>
            </ipv4>
          </interface>
        </interfaces>
      </config>
    </edit-config>
</rpc>
]]>]]>
```

Поскольку этот XML-документ создан в текстовом редакторе, его можно с легкостью скопировать и вставить в терминал активного сеанса NETCONF.

После вставки документа в терминал интерактивного сеанса NETCONF вы увидите ответное сообщение об успешном завершении операции, как показано ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <ok/>
</rpc-reply>
]]>]]>
```



Об этом мы говорили уже несколько раз, но повторим снова, потому что это чрезвычайно важно. Когда вы начинаете работать с интерфейсами NETCONF API (или с любыми RESTful API), необходимо твердо знать, как сформировать правильный объект для запроса. Зачастую это становится трудной задачей для начинающих, но они могут надеяться на существование простых способов, помогающих освоить создание таких объектов. Помощь можно получить из документации на API, от инструментальных средств, компонующих интерфейс с помощью файлов определений схемы более низкого уровня, таких как XSD или модули YANG, в конце концов, полезными могут оказаться даже текстовые команды CLI на целевом устройстве. Например, на устройствах Cisco Nexus и Juniper Junos имеются команды CLI, которые точно показывают пользователю, какой XML-документ необходим для выполнения требуемого запроса. Вскоре мы рассмотрим соответствующие примеры.

После общего обзора и рассмотрения некоторых практических приемов работы с API на основе протокола HTTP и NETCONF вы должны хорошо понимать, как можно автоматизировать сетевые устройства с помощью этих API. Теперь мы сосредоточим внимание на применении языка программирования Python для автоматизации устройств с использованием API на основе HTTP, NETCONF и SSH.

Автоматизация с использованием сетевых API

Как уже было отмечено выше, существуют различия между инструментальными средствами, предназначенными для исследования и изучения практического применения API, и инструментальными средствами, ориентированными на «потребление» (consume) API, которые в большей степени соответствуют

Tlgm: @it_boooks

реальной эксплуатационной функциональной модели. До настоящего момента мы рассматривали cURL и Postman с точки зрения исследования и изучения API на основе HTTP, а интерактивный интерфейс NETCONF поверх сеанса SSH как средство изучения стека протоколов NETCONF. В этом разделе главы мы рассмотрим использование языка программирования Python для автоматизации сетевых устройств. Основное внимание будет сосредоточено на трех специализированных библиотеках Python:

- requests – широко распространенная и удобная для пользователя библиотека поддержки протокола HTTP на языке Python. Эту библиотеку мы используем для автоматизации устройств и контроллеров как для RESTful API на основе HTTP, так и для API на основе HTTP, не поддерживающих концепцию REST;
- ncclient – это клиент NETCONF на языке Python. Эту библиотеку мы будем применять для автоматизации устройств с поддержкой NETCONF;
- netmiko – основной сетевой SSH-клиент на языке Python. Этой библиотекой мы воспользуемся для автоматизации устройств, предоставляющих поддержку протокола SSH непосредственно на самом целевом устройстве без программных API.

Начнем изучение с библиотеки requests и ее практического применения для обмена данными с API на основе протокола HTTP.

Использование библиотеки requests

Вы уже видели, как создаются вызовы API на основе протокола HTTP из командной строки с помощью утилиты cURL, а также с использованием графического пользовательского интерфейса приложения Postman. Это великолепные механизмы для изучения применения конкретного API, но в реальной практике для написания скрипта или программы, действительно полезной для автоматизации сетевых устройств, вы должны уметь самостоятельно формировать вызовы API вручную из любого скрипта или программы. В текущем разделе подробно рассматривается библиотека requests на языке Python, существенно упрощающая работу с веб-интерфейсами API. Здесь мы приводим обзоры различных API, таких как Arista eAPI, Cisco ASA RESTful API, Cisco NX-API и Cisco IOS-XE RESTCONF API, тем не менее рекомендуется изучить этот раздел от начала до конца, так как главное внимание все же сосредоточено на практическом использовании библиотеки requests.



Чтобы установить библиотеку requests, воспользуйтесь утилитой `pip`:

```
[sudo] pip install requests
```

Сразу переходим к первому примеру с применением библиотеки requests. Это полнофункциональный скрипт на языке Python, позволяющий получить конфигурацию интерфейса на устройстве CISCO ASA. Здесь выполняется тот же запрос GET, который рассматривался в примерах с использованием cURL и Postman.

Tlgm: @it_boooks


```
#!/usr/bin/env python

import json
import requests
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    }
    url = "https://asav/api/interfaces/physical"
    response = requests.get(url, auth=auth, headers=headers, verify=False)
```

Ниже приведен тот же скрипт с подробными комментариями, позволяющими понять, что именно делает каждая его строка.

```
#!/usr/bin/env python

# Импортируется модуль json, чтобы обеспечить возможность кодирования и декодирования
# JSON-объектов, передаваемых по сети. Несмотря на то что в языке Python мы работаем
# с JSON-объектами как со словарями, в сетевой среде они передаются как JSON-строки
# в вызовах API.
# Таким образом, необходим способ преобразования словарей в строки, понятные сетевому
# устройству (веб-серверу). Также необходимо преобразование в обратном направлении.
# При получении ответа от устройства, использующего кодировку JSON, полученная строка
# должна быть преобразована в словарь, чтобы с ней можно было работать средствами языка
# Python. Для выполнения этих операций предназначен модуль json.
import json

# Библиотека requests (Python) предназначена для работы с системами, основанными на
протоколе HTTP.
# Также используется вспомогательная функция из библиотеки requests
# с именем `HTTPBasicAuth` для упрощения процедуры аутентификации
import requests
from requests.auth import HTTPBasicAuth

# Скрипт выполняется как программа, запускаемая напрямую.
if __name__ == "__main__":

    # Создается объект аутентификации, использующий вспомогательную функцию
    # HTTPBasicAuth. Это работает, если устройство поддерживает основные
    # возможности аутентификации. Отметим, что имя переменной auth выбрано
    # произвольно, как, впрочем, и все остальные переменные. Здесь они просто
    # соответствуют именам параметров, используемым в библиотеке requests.
    # Если вспомогательная функция HTTPBasicAuth не используется,
    # то можно определить аутентификационные данные в функции get с
    # использованием кортежа, например auth=(ntc, ntc123).
    auth = HTTPBasicAuth('ntc', 'ntc123')

    # Эта команда создает словарь Python для заголовков HTTP-запроса,
    # которые будут использоваться для вызовов API. Здесь создаются
    # два заголовка: Content-Type и Accept. Это те же самые заголовки,
    # которые рассматривались в примере с использованием Postman.
    headers = {
```

Tlgm: @it_boooks


```
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    }

    # URL сохраняется как переменная с именем url для обеспечения модульности
    # кода и упрощения следующей команды.
    url = "https://asav/api/interfaces/physical"

    # Последняя команда - вызов API, выполняемый с использованием запросов.
    # В библиотеке requests существуют функции для каждого ключевого слова HTTP,
    # а в этом примере выполняется запрос GET, следовательно, применяется
    # функция `get`, в которую передаются четыре объекта.
    # Первый передаваемый объект обязательный - это URL, а остальные должны быть
    # ключевыми словами (парами ключ-значение). Здесь используются три ключевых слова
    # как аргументы с именами auth, headers и verify. Значения ключевых слов
    # auth и headers устанавливаются равными значениям ранее созданных переменных,
    # а для значения verify определяется значение False, так как устройство Cisco ASA
    # использует самоподписанный сертификат, поэтому мы не можем его проверить.
    response = requests.get(url, auth=auth, headers=headers, verify=False)
```



Если выполнить этот скрипт для устройства, использующего самоподписанный сертификат или HTTPS-соединение без проверки, то в ответ будет получено предупреждающее сообщение. При использовании библиотеки requests можно отключить вывод этого сообщения при помощи следующей команды Python:

```
requests.packages.urllib3.disable_warnings()
```

Продолжим расширение возможностей скрипта на основе приведенного выше примера. Теперь попробуем обновить описание интерфейса с помощью тех же запросов, которые были продемонстрированы в примере с использованием приложения Postman.

Для внесения изменений в конфигурацию с помощью библиотеки requests необходимо сделать точно такие же три корректировки, как в примере с Postman: обновить URL, обновить тип запроса HTTP и отправить данные в теле запроса.

```
payload = {
  "kind": "object#GigabitInterface",
  "interfaceDesc": "Configured by Python"
}
url = "https://asav/api/interfaces/physical/GigabitEthernet0_API_SLASH_0"
response = requests.patch(url, data=json.dumps(payload), auth=auth,
                          headers=headers, verify=False)
```



Обратите особое внимание на используемые ключевые слова HTTP. Рассматриваемый в приведенном примере запрос использует функцию `patch()` в применении к обновляемому ресурсу. Если бы ресурс создавался, следовало бы воспользоваться функцией `post()`, а при замене ресурса потребовалась бы функция `put()`. Все эти функции будут рассматриваться в следующих примерах текущей главы.

Обновление URL и типа запроса – это простые изменения. В нашем примере переменная `url` обновляется в предпоследней команде, после которой вызывается функция `patch()`.

Tlgm: @it_boooks

Теперь сосредоточимся на том, как передаются данные в теле HTTP-запроса. Именно здесь необходимо четко осознавать различие между словарем Python и строкой JSON. При формировании требуемого тела запроса мы работаем со словарями языка Python, но в сетевой среде запросы передаются в форме строки JSON. Чтобы преобразовать словарь в корректную строку JSON, используется функция `dumps()` из модуля `json`. Эта функция принимает словарь как аргумент и выполняет его преобразование в строку JSON. Далее сформированный объект типа строка передается через сетевую среду с присваиванием данных ключу `data` с помощью функции `patch()`.

После ознакомления с основами использования библиотеки requests мы продолжим ее освоение и рассмотрим еще три API на основе протокола HTTP. Еще раз напомним, что хотя мы и рассматриваем в этой главе различные API, рекомендуется читать ее без пропусков и помнить о том, что это не документация на какой-либо конкретный API.

Основы использования Cisco NX-API

Рассмотрим Cisco Nexus NX-API и одновременно продолжим изучение возможностей библиотеки requests. При этом необходимо помнить о следующих фактах:

- интерфейс NX-API не является RESTful API на основе HTTP. Другими словами, это API на основе HTTP, который не соблюдает всех принципов REST. Вне зависимости от того, какую операцию нужно выполнить, всегда используется HTTP-запрос POST. Даже при выполнении команды `show` применяется запрос POST;
- в запросе POST данные обязательно должны передаваться как «полезная нагрузка» (payload) в теле запроса. Здесь весьма важное значение имеет использование надежных инструментальных средств API и внимательное изучение документации. В коммутаторах Nexus имеется встроенное инструментальное средство NX-API Developer Sandbox, которое будет рассмотрено более подробно, чтобы понять требуемую структуру объекта payload;
- для вызовов NX-API всегда используется следующий формат URL: *http(s)://<ip-address-nexus>/ins*.

Использование NX-API Developer Sandbox Прежде чем приступить к работе с NX-API средствами языка Python, необходимо уделить некоторое внимание инструменту NX-API Developer Sandbox, который поможет нам точно узнать правильную структуру HTTP-запроса при использовании NX-API.

Если использование NX-API разрешено, то можно изучать коммутатор Nexus с помощью веб-браузера. После регистрации (входа) на устройстве вы увидите пользовательский веб-интерфейс NX-API Developer Sandbox, показанный на рис. 7.8.

NX-API Developer Sandbox – это встроенный (on-box) инструмент, позволяющий тестировать API, а также понять, как должны выглядеть объекты за-

Tlgm: @it_boooks

проса и ответа. При этом не требуется писать какой-либо программный код. Это похоже на сервис, предлагаемый приложением Postman, но в данном случае «песочница» для разработчиков существует непосредственно на каждом коммутаторе Nexus.



Упомянутое выше встроенное инструментальное средство (on-box tool) представляет собой инструментальное средство, размещенное непосредственно на сетевом устройстве. В частности, NX-API Developer Sandbox – это веб-утилита, размещенная на каждом коммутаторе Nexus.

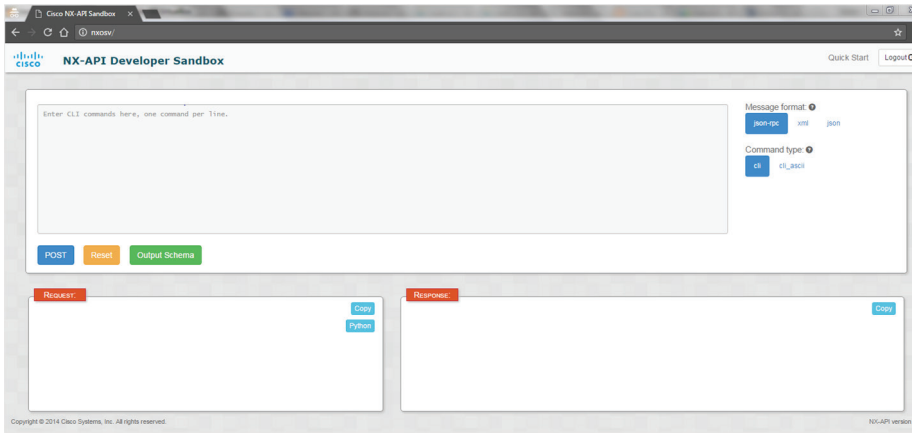


Рис. 7.8 ❖ Пользовательский интерфейс Cisco NX-API Developer Sandbox

Сразу после ввода команды в текстовой панели, располагающейся в верхней левой части экрана, автоматически генерируется объект запроса в формате JSON в панели, находящейся внизу слева (см. рис. 7.9).

После щелчка по синей кнопке POST выполняется вызов API на коммутаторе, а HTTP-ответ выводится в панели, размещенной внизу справа. Ответ представляет собой тот же объект, который мы получим при использовании средств языка Python для выполнения вызовов API.

Извлечем объект запроса и сохраним его в переменной, чтобы можно было воспользоваться им в скрипте на языке Python.

```
payload = {
    "ins_api": {
        "version": "1.0",
        "type": "cli_show",
        "chunk": "0",
        "sid": "1",
        "input": 'show version',
        "output_format": "json"
    }
}
```

Tlgm: @it_boooks

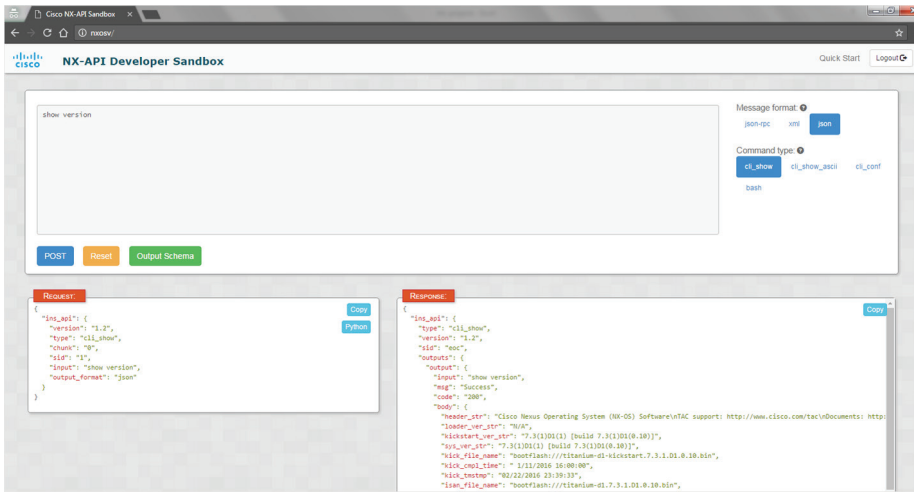


Рис. 7.9 ❖ Формирование запроса в формате JSON в NX-API Developer Sandbox

Напомним, что объект `payload` – это словарь, поэтому необходимо преобразовать его в строку перед передачей на устройство. Преобразование выполняется с помощью функции `dumps()` из Python-модуля `json`. Кроме того, следует помнить, что каждый запрос через NX-API представляет собой запрос POST, что однозначно определяет функцию библиотеки requests, которую мы должны использовать.

Использование NX-API в скрипте на языке Python Рассмотрим полнофункциональный скрипт на языке Python, реализующий вызов API для выполнения команды `show version`.

```
import json
import requests
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'application/json'
    }
    url = 'http://nxos-spine1/ins'

    payload = {
        "ins_api": {
            "version": "1.0",
            "type": "cli_show",
            "chunk": "0",
            "sid": "1",
```

Tlgm: @it_boooks


```
            "input": 'show version',
            "output_format": "json"
        }
    }

    response = requests.post(url, data=json.dumps(payload),
                             headers=headers, auth=auth)
    print(response)
```

Практически все элементы приведенного выше скрипта так или иначе обсуждались ранее, но краткое напоминание не будет лишним.

Здесь используется модуль `json`, с его помощью выполняется сериализация и десериализация объектов (строк) JSON в словари и обратно. Импортируется библиотека requests, поскольку она необходима для формирования HTTP-запросов к устройствам Nexus. Также импортируется объект `HTTPBasicAuth`, используемый для упрощения процедуры аутентификации.

В первой части скрипта выполняется параметризация регистрационных данных, заголовков и URL, необходимых для обращения к NX-API. Здесь не наблюдается никаких отличий от того, что мы делали при использовании ASA RESTful API.

```
auth = HTTPBasicAuth('ntc', 'ntc123')
headers = {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
}
url = 'http://nxos-spine1/ins'
```

Можно видеть, что регистрационные данные не отличаются от тех, которые использовались бы при входе на устройство Nexus через сеанс SSH (если предположить, что установлены соответствующие права доступа).

Обратите внимание на URL. Это строго фиксированный URL для всех соединений NX-API. Он состоит из полного имени домена IP/FQDN коммутатора и суффикса `/ins`, представляющего собой сокращение в виде первых трех букв названия Insieme, компании, включенной в состав корпорации Cisco в 2012 году для обеспечения формирования сетевой среды для центров данных и создания SDN-решений.

Осталось рассмотреть завершающую часть скрипта:

```
payload = {
    "ins_api": {
        "version": "1.0",
        "type": "cli_show",
        "chunk": "0",
        "sid": "1",
        "input": 'show version',
        "output_format": "json"
    }
}
```

Tlgm: @it_boooks


```
response = requests.post(url, data=json.dumps(payload),
                         headers=headers, auth=auth)
print(response)
```

Здесь можно видеть, что содержимое переменной `payload` взято непосредственно из NX-API Developer Sandbox, а в среде Python эта переменная представляет собой словарь, состоящий из одной пары ключ-значение. Ключом является `ins_api`, а значением – вложенный словарь с шестью парами ключ-значение.

Все описанные выше объекты объединяются в следующей строке, в которой выполняется вызов API на коммутаторе Nexus. Обратите внимание на синтаксис команды Python `requests.post()` – имя функции соответствует ключевому слову, то есть типу требуемого HTTP-запроса. Например, если бы нужно было выполнить запрос GET, то команда должна была бы выглядеть так: `requests.get()`.

В функцию post() из библиотеки requests передаются четыре объекта языка Python. Первый аргумент обязателен и должен располагаться именно в этой позиции (позиционный аргумент) – это URL. Следующие три аргумента передаются в форме ключевых слов, требуемых функцией `post()`, – это `data`, `headers` и `auth`.

Как уже отмечалось выше, не следует забывать о выполнении преобразования `data=json.dumps(<dict>)`, так как необходимо сериализовать словарь в формат строки для передачи в коммутатор Nexus.

Самая последняя строка скрипта просто выводит полученный ответ.

Если сохранить этот скрипт в файле *\nxapi-cli.py* и запустить его из командной строки, то получим следующий результат:

```
ntc@ntc:~$ python nxapi-cli.py
<Response [200]>
ntc@ntc:~$
```

Получен ответ – HTTP-код 200, то есть все работает, как предполагалось, но возникает естественный вопрос: а где же данные, которые мы хотели бы увидеть после выполнения команды `show version`?

Использование NX-API в интерактивном интерпретаторе Python Выполним еще раз этот скрипт с использованием флага `-i` в командной строке, чтобы автоматически перейти в интерактивный интерпретатор языка Python, но сохранить возможность доступа ко всем объектам выполняемого скрипта. Действие флага `-i` было объяснено в главе 4. Это очень удобный способ тестирования и устранения проблем.

```
ntc@ntc:~$ python -i nxapi-cli.py
<Response [200]>
>>>
```

Теперь мы находимся в среде интерактивного интерпретатора Python и можем увидеть объекты из выполненного скрипта с помощью функции `dir()`, которая также была описана в главе 4.

Tlgm: @it_boooks


```
>>> dir()
['HTTPBasicAuth', '__builtins__', '__doc__', '__name__', '__package__',
'auth', 'json', 'payload', 'requests', 'response', 'url']
>>>
```

Продолжим исследование и воспользуемся функцией `dir()` для объекта `response`, поскольку эта функция выводит все атрибуты и методы заданного объекта.

```
>>> dir(response)
['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__',
'__doc__', '__format__', '__getattribute__', '__getstate__', '__hash__',
'__init__', '__iter__', '__module__', '__new__', '__nonzero__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__',
'__str__',
'__subclasshook__', '__weakref__', '_content', '_content_consumed',
'apparent_encoding', 'close', 'connection', 'content', 'cookies',
'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect',
'is_redirect', 'iter_content', 'iter_lines', 'json', 'links',
'ok', 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text',
'url']
>>>
```

Рассмотрим подробнее два атрибута – `status_code` и `text`.

Атрибут `status_code` предоставляет доступ к HTTP-коду ответа в формате целого числа.

```
>>> print(response.status_code)
200
>>> type(response.status_code)
<type 'int'>
>>>
```

В атрибуте `text` хранятся реальные данные ответа от коммутатора Nexus. Это те же самые данные, которые мы видели в окне ответа (в нижнем правом углу) NX-API Developer Sandbox.

```
>>> print(response.text)
{
  "ins_api":  {
    "type": "cli_show",
    "version":  "1.2",
    "sid":  "eoc",
    "outputs":  {
      "output": {
        "input":  "show version",
        "msg":  "Success",
        "code": "200",
        "body": {
          "header_str": "Cisco Nexus Operating System (NX-OS) Software...",
          "loader_ver_str": "N/A",
          "kickstart_ver_str":  "7.3(1)D1(1) [build 7.3(1)D1(0.10)]",
          "sys_ver_str":  "7.3(1)D1(1) [build 7.3(1)D1(0.10)]",
```

Tlgm: @it_boooks


```
          "kick_file_name": "bootflash:///titanium-d1-kickstart.7.3.1.D1.0...",
          "kick_cmpl_time": " 1/11/2016 16:00:00",
          "kick_tmstmp":  "02/22/2016 23:39:33",
          "isan_file_name": "bootflash:///titanium-d1.7.3.1.D1.0.10.bin",
          "isan_cmpl_time": " 1/11/2016 16:00:00",
          "isan_tmstmp":  "02/23/2016 01:43:36",
          "chassis_id": "NX-OSv Chassis",
          "module_id":  "NX-OSv Supervisor Module",
          "cpu_name": "Intel(R) Xeon(R) CPU @ 2.50G",
          "memory": 4002312,
          "mem_type": "kB",
          "proc_board_id":  "TM604E634FB",
          "host_name":  "nxos-spine1",
          "bootflash_size": 1582402,
          "kern_uptm_days": 0,
          "kern_uptm_hrs":  3,
          "kern_uptm_mins": 16,
          "kern_uptm_secs": 45,
          "manufacturer": "Cisco Systems, Inc."
        }
      }
    }
  }
}
>>>
```

Сохраним эти данные в переменной и извлечем из них значение ключа `host_name`.

```
>>> result = response.text
>>>
>>> print(result['ins_api']['outputs']['output']['body']['host_name'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string indices must be integers
>>>
```

Что произошло, и по какой причине возникла ошибка?

Обработка ответа в формате JSON от API на основе протокола HTTP Ранее уже неоднократно подчеркивалось, что в сетевой среде передаются строки в формате JSON. Теперь это можно подтвердить на практике, проверив тип данных объекта `result`.

В предыдущих примерах использовалась функция `json.dumps()` для преобразования словаря в строку JSON. Но теперь необходимо выполнить обратную операцию: строку в формате JSON нужно преобразовать в словарь. Для этого используется функция `json.loads()`, которая перекодирует JSON-строку в словарь.

```
>>> result_dict = json.loads(result)
>>>
>>> type(result_dict)
<type 'dict'>
```

Tlgm: @it_boooks


```
>>>
>>> print(result_dict['ins_api']['outputs']['output']['body']['host_name'])
nxos-spine1
>>>
```

Главный урок, который можно извлечь из двух последних примеров: не следует забывать о необходимых процедурах сериализации и десериализации словарей в строки и наоборот и всегда проверять типы данных. В последнем примере наглядно показано, что это легко сделать с помощью встроенной функции `type()`.

Дополнительные примеры использования NX-API в скриптах на языке Python Теперь необходимо выйти из интерактивного интерпретатора Python и внести в скрипт некоторые изменения, которые сделают более универсальным его использование. Сначала добавляется возможность передачи в скрипт произвольной команды и формата сообщения NX-API.

```
import json
import sys
import requests
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    url = 'http://nxos-spine1/ins'
    command = sys.argv[1]

    if len(sys.argv) > 2:
        command_type = sys.argv[2]
    else:
        command_type = 'cli_show'

    payload = {
        "ins_api": {
            "version": "1.0",
            "type": command_type,
            "chunk": "0",
            "sid": "1",
            "input": command,
            "output_format": "json"
        }
    }

    response = requests.post(url, data=json.dumps(payload),  auth=auth)

    print('STATUS CODE: ' + response.status_code)

    print('RESPONSE:')
    results = json.loads(response.text)
    print(json.dumps(results, indent=4))
```

В этом примере следует обратить особое внимание на следующие изменения:

- ключи `type` и `input` в словаре теперь параметризованы. Это особые ключи для NX-API. Сейчас они стали переменными, в которые можно передать данные из терминала при выполнении скрипта;

Tlgm: @it_boooks

- напомним, что объект sys.argv рассматривался в главе 4. В этом примере он необходим для передачи как минимум одного аргумента – команды или argv[1]. Первый аргумент argv[0] всегда содержит имя самого скрипта;
- пользователь также может передать или изменить необязательный аргумент type (представленный argv[2]) в зависимости от выполняемых команд или от указанного в ответе типа. В NX-API Developer Sandbox это можно протестировать в визуальном представлении, но для скриптов корректный тип type устанавливается значениями cli_show (для возврата строки в формате JSON), cli_show_ascii (для возврата обычного текста без форматирования) и cli_conf (для установки режима передачи конфигурационных команд);
- теперь выводится значение кода состояния status_code и содержимое атрибута text из ответа.

Передача команды show в скрипт Выполним измененный скрипт с передачей в него команды show version в качестве аргумента.

```
ntc@ntc:~$ python nxapi-cli.py "show version"
STATUS CODE:  200
RESPONSE:
{
    "ins_api": {
        "outputs": {
            "output": {
                "msg": "Success",
                "input": "show version",
                "code": "200",
                "body": {
                    "kern_uptm_secs": 34,
                    "kick_file_name": "bootflash:///titanium-d1-kickstart.7.3.1.D1.0.10.bin",
                    "loader_ver_str": "N/A",
                    "module_id": "NX-OSv Supervisor Module",
                    "kick_tmstmp": "02/22/2016 23:39:33",
                    "isan_file_name": "bootflash:///titanium-d1.7.3.1.D1.0.10.bin",
                    "sys_ver_str": "7.3(1)D1(1) [build 7.3(1)D1(0.10)]",
                    "bootflash_size": 1582402,
                    "kickstart_ver_str": "7.3(1)D1(1) [build 7.3(1)D1(0.10)]",
                    "kick_cmpl_time": " 1/11/2016 16:00:00",
                    "chassis_id": "NX-OSv Chassis",
                    "proc_board_id": "TM604E634FB",
                    "memory": 4002312,
                    "kern_uptm_mins": 10,
                    "cpu_name": "Intel(R) Xeon(R) CPU @ 2.50G",
                    "kern_uptm_hrs": 4,
                    "isan_tmstmp": "02/23/2016 01:43:36",
                    "manufacturer": "Cisco Systems, Inc.",
                    "header_str": "Cisco Nexus Operating System (NX-OS) Software\nTAC
support: ...",
                    "isan_cmpl_time": " 1/11/2016 16:00:00",
                    "host_name": "nxos-spine1",
```

Tlgm: @it_boooks


```
                    "mem_type": "kB",
                    "kern_uptm_days": 0
                }
            }
        },
        "version": "1.2",
        "type": "cli_show",
        "sid": "eoc"
    }
}
ntc@ntc:~$
```

Поскольку формат ответа можно изменить на обычный неформатированный текст, теперь можно передать дополнительный параметр, установив для него значение `cli_show_ascii`. Это еще один параметр, использование которого можно изучить в графической среде NX-API Developer Sandbox.

```
ntc@ntc:~$ python nxapi-cli.py "show version" "cli_show_ascii"
STATUS CODE:  200
RESPONSE:
{
    "ins_api": {
        "outputs": {
            "output": {
                "msg": "Success",
                "input": "show version",
                "code": "200",
                "body": "Cisco Nexus Operating System (NX-OS) Software\nTAC support: ..."
            }
        },
        "version": "1.2",
        "type": "cli_show_ascii",
        "sid": "eoc"
    }
}
ntc@ntc:~$
```

После установки типа `cli_show_ascii` вы получите ответ в виде строки неструктурированных данных, как если бы выполнили команду из интерфейса CLI, а ответ был возвращен в виде значения ключа `body`.

Передача команд конфигурации в скрипт Если продолжить тестирование различных свойств в среде NX-API Developer Sandbox, то можно заметить, что при отправке параметров конфигурации они передаются как строки с разделителем в виде точки с запятой между каждой командой, включенной в содержимое ключа `input` при использовании формата сообщения `json`.

Воспользуемся тем же самым скриптом, чтобы показать это на примере.

```
ntc@ntc:~$ python nxapi-cli.py "vlan 10 ; vlan 20 ; exit ;" "cli_conf"

STATUS CODE:  200
RESPONSE:
```

Tlgm: @it_boooks


```
{
    "ins_api": {
        "outputs": {
            "output": [
                {
                    "msg": "Success",
                    "body": {},
                    "code": "200"
                },
                {
                    "msg": "Success",
                    "body": {},
                    "code": "200"
                },
                {
                    "msg": "Success",
                    "body": {},
                    "code": "200"
                }
            ]
        },
        "version": "1.2",
        "type": "cli_conf",
        "sid": "eoc"
    }
}
ntc@ntc:~$
```

При внесении изменений в конфигурацию с помощью NX-API вы получаете отдельный элемент ответа на каждую отправленную команду. Отметим, что ответ в предыдущем примере содержит отдельный словарь для каждой успешно выполненной команды в составе ключа `output`.

Итак, мы подробно рассмотрели практическое использование библиотеки requests для выполнения вызовов API на устройствах Cisco ASA и Nexus. Теперь вам хорошо знакомы некоторые типовые шаблоны операций, выполняемых на этих устройствах. Но вы узнаете гораздо больше при использовании других API.

Общее описание Arista eAPI

Теперь рассмотрим Arista eAPI, программный интерфейс, который очень похож на Cisco NX-OS NX-API. При изучении приводимых ниже примеров использования eAPI необходимо помнить о следующих фактах:

- eAPI – интерфейс на основе протокола HTTP, не поддерживающий концепции REST. Другими словами, это API на основе HTTP, который не соблюдает всех принципов REST. Вне зависимости от типа выполняемой операции всегда используется HTTP-запрос POST, то есть даже если необходимо выполнить команду `show`, используется запрос POST;
- напомним, что запросы POST требуют передачи данных как «полезной нагрузки» (payload) в теле запроса. Здесь весьма важное значение имеет

Tlgm: @it_boooks

использование надежных инструментальных средств API и внимательное изучение документации. В коммутаторах Arista имеется встроенное инструментальное средство Command Explorer, которое будет рассмотрено более подробно, чтобы понять требуемую структуру объекта payload;

- для вызовов eAPI всегда используется следующий формат URL: *http(s)://<ip-address-eos>/command-api*.

Использование инструмента Command Explorer интерфейса eAPI Прежде чем приступить к работе с eAPI средствами языка Python, необходимо уделить некоторое внимание инструменту Command Explorer, который поможет нам точно узнать правильную структуру HTTP-запроса при использовании eAPI. Если использование eAPI разрешено, то можно изучать коммутатор Arista с помощью веб-браузера. После регистрации (входа) на устройстве вы увидите пользовательский веб-интерфейс Command Explorer, показанный на рис. 7.10.

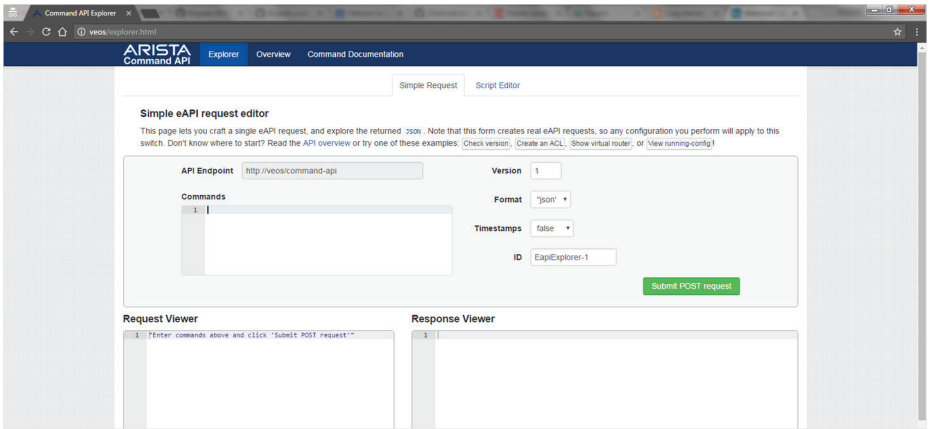


Рис. 7.10 ❖ Пользовательский интерфейс Arista eAPI Command Explorer

Подобно утилите Cisco NX-API Developer Sandbox, программа Arista eAPI Command Explorer представляет собой встроенное инструментальное средство (on-box tool), позволяющее тестировать API, а также понять, как должны выглядеть объекты запроса и ответа. При этом не требуется писать какой-либо программный код.

Сразу после ввода команд в список Commands в центральной панели нужно щелкнуть по зеленой кнопке **Submit POST request** (Выполнить запрос POST), чтобы обратиться к функциям API. Затем можно просмотреть объекты запроса и ответа в панелях, расположенных внизу слева и внизу справа соответственно.

Рассмотрим пример выполнения команды `show version` на коммутаторе Arista (см. рис. 7.11).

Tlgm: @it_boooks

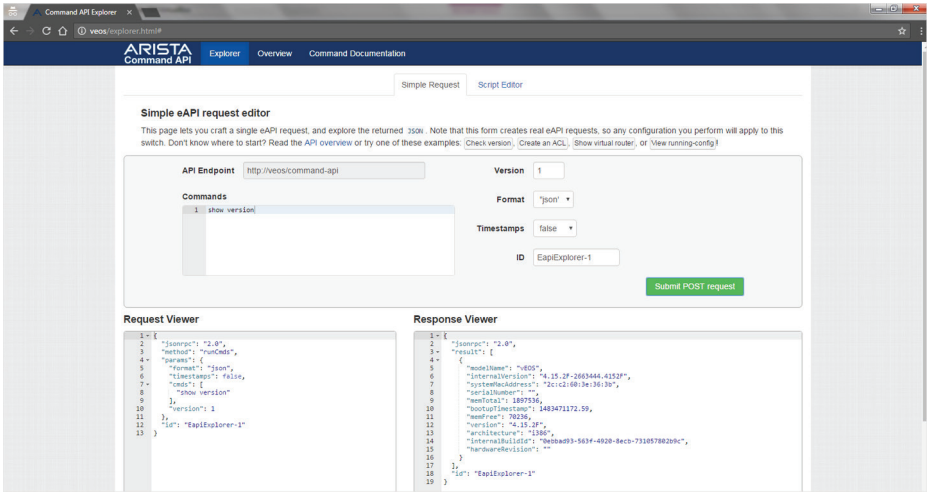


Рис. 7.11 ❖ Запрос и ответ в формате JSON в программе Arista eAPI Command Explorer

Объект ответа, показанный в программе Command Explorer, представляет собой тот же самый объект, который в дальнейшем будет получен при вызове API средствами языка Python.

Обратите внимание на то, что объект JSON, отправляемый на устройство Arista, отличается от объекта, который передавался на устройство Cisco Nexus. Это вполне обычный случай при использовании различных API, но здесь требуется особое внимание к типам данных передаваемых команд.

Использование eAPI в скрипте на языке Python Рассмотрим скрипт на языке Python, в котором используется библиотека requests для обмена данными с eAPI. В скрипте выполняется команда `show vlan brief`, и ответ на нее выводится в терминале вместе с HTTP-кодом состояния.

```
import json
import sys
import requests
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    url = 'http://eos-spine1/command-api'

    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "format": "json",
            "timestamps": False,
            "cmds": [
```

Tlgm: @it_boooks


```
                "show vlan brief"
            ],
            "version": 1
        },
        "id": "EapiExplorer-1"
    }

    response = requests.post(url, data=json.dumps(payload),  auth=auth)
    print('STATUS CODE: ' + response.status_code)

    print('RESPONSE:')
    results = json.loads(response.text)
    print(json.dumps(results, indent=4))
```

Скрипт сохраняется в файле *eapi-requests.py* и при выполнении выводит следующий результат:

```
ntc@ntc:~$ python eapi-requests.py
STATUS CODE:  200
RESPONSE:
{
    "jsonrpc": "2.0",
    "result": [
        {
            "sourceDetail": "",
            "vlans": {
                "1": {
                    "status": "active",
                    "interfaces": {},
                    "dynamic": false,
                    "name": "default"
                },
                "30": {
                    "status": "active",
                    "interfaces": {},
                    "dynamic": false,
                    "name": "VLAN0030"
                },
                "20": {
                    "status": "active",
                    "interfaces": {},
                    "dynamic": false,
                    "name": "VLAN0020"
                }
            }
        }
    ],
    "id": "EapiExplorer-1"
}
ntc@ntc:~$
```

Здесь можно видеть, что ответ представляет собой объект JSON с вложенными структурами. Самый важный ключ – `result`, который является списком сло-

Tlgm: @it_boooks

варей. Для каждой выполненной команды создан отдельный элемент списка (словарь). Поскольку была передана только одна команда, в ответе содержится список с одним элементом.

Оптимизация скрипта, использующего eAPI Функциональность приведенного выше скрипта можно немного улучшить, если вспомнить приемы, рассмотренные в главе 4. Скрипт будет выводить только идентификатор VLAN ID и имя каждой виртуальной сети VLAN, сконфигурированной в системе. Таким образом, необходимо получить следующий результат:

```
VLAN ID    NAME
1          default
20         VLAN0020
30         VLAN0030
```

Эти изменения в большей степени касаются использования языка Python, чем работы непосредственно с API, но и такой пример все же дает полезные практические навыки. Мы сохраним результат, извлечем из него словарь VLAN, затем организуем проход по словарю в цикле с выводом на экран терминала каждого найденного имени VLAN.

```
response = requests.post(url, data=json.dumps(payload), auth=auth)
rsp = json.loads(response.text)
vlans = rsp['result'][0]['vlans']
print('{:12}{:<10}'.format('VLAN ID', 'NAME'))
for vlan_id, config in vlans.items():
    print('{:<12}{:<10}'.format(vlan_id,  config['name']))
```

После запуска вновь созданного скрипта будет выведен следующий результат:

```
ntc@ntc:~$ python eapi-requests.py
VLAN ID     NAME
1           default
30          VLAN0030
20          VLAN0020
ntc@ntc:~$
```

Использование eAPI для автоматической конфигурации описаний интерфейсов на основе данных LLDP Продолжим изучение практического применения eAPI и попробуем создать что-то немного более полезное. Напишем скрипт на языке Python, выполняющий автоматическую конфигурацию описаний Ethernet-интерфейсов на основе данных LLDP, полученных от соседних устройств, для двух коммутаторов Arista, являющихся основными в локальной сети.

Для этого необходимо сделать скрипт модульным, чтобы обеспечить поддержку нескольких устройств и организовать простой способ отправки нескольких вызовов API без формирования нескольких различных объектов payload в скрипте.

Главная цель – выполнение автоматической конфигурации описаний интерфейсов, чтобы они выглядели следующим образом:

Tlgm: @it_boooks


```
interface Ethernet1
   description Connects to interface Ethernet1 on neighbor eos-spine2.ntc.com
   (описание Устанавливает соединение с интерфейсом Ethernet1 на соседнем устройстве
eos-spine2.ntc.com)
   no switchport
!
interface Ethernet2
   description Connects to interface Ethernet2 on neighbor eos-spine2.ntc.com
   (описание Устанавливает соединение с интерфейсом Ethernet2 на соседнем устройстве
eos-spine2.ntc.com)
   no switchport
!
```

В качестве самостоятельного упражнения читателю предлагается внимательно изучить приведенный ниже скрипт. Отметим только, что это всего лишь одна функция, выполняющая запрос eAPI на коммутаторе Arista EOS. Все тонкости программирования на языке Python расматривались в главе 4. Методика написания модульного кода с целью упрощения его многократного использования в дальнейшем в различных скриптах со временем становится все более востребованной.

```
import json
import sys
import requests
from requests.auth import HTTPBasicAuth

def issue_request(device, commands):
    """Make API request to EOS device returning JSON response
       (Создание запроса API к устройству EOS, возвращающему ответ в формате JSON)
    """

    auth = HTTPBasicAuth('ntc', 'ntc123')
    url = 'http://{}/command-api'.format(device)
    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "format": "json",
            "timestamps": False,
            "cmds": commands,
            "version": 1
        },
        "id": "EapiExplorer-1"
    }

    response = requests.post(url, data=json.dumps(payload),  auth=auth)
    return json.loads(response.text)

def get_lldp_neighbors(device):
    """Get list of neighbors
       (Получение списка соседних устройств
    Sample response for a single neighbor:
```

Tlgm: @it_boooks


```
    (Пример ответа от одного из соседних устройств:)
        {
          "ttl": 120,
          "neighborDevice": "eos-spine2.ntc.com",
          "neighborPort": "Ethernet2",
          "port": "Ethernet2"
        }
    """
    commands = ['show lldp neighbors']
    response = issue_request(device, commands)
    neighbors = response['result'][0]['lldpNeighbors']
    # соседние устройства (имена) возвращаются в виде списка словарей
    return neighbors

def configure_interfaces(device, neighbors):
    """Configure interfaces in a single API call per device
       (Конфигурирование интерфейсов в одном вызове API для каждого устройства)
    """
    command_list = ['enable', 'configure']
    for neighbor in neighbors:
        local_interface = neighbor['port']
        if local_interface.startswith('Eth'):
            # Проход с учетом существующих соседних устройств
            description = 'Connects to interface {} on neighbor {}'.format(
                neighbor['neighborPort'],
                neighbor['neighborDevice'])
            description = 'description ' + description
            interface = 'interface {}'.format(local_interface)
            cmds = [interface, description]
            command_list.extend(cmds)
    response = issue_request(device, command_list)

if __name__ == "__main__":
    # Имена устройств являются полными доменными именами FQDN
    devices = ['eos-spine1', 'eos-spine2']
    for device in devices:
        neighbors = get_lldp_neighbors(device)
        configure_interfaces(device, neighbors)
        print('Auto-Configured Interfaces for {}'.format(device))
```

После внимательного изучения приведенных выше примеров использования трех различных API вы наверняка почувствовали бóльшую уверенность при практическом применении библиотеки requests. В этих примерах использовался встроенный RESTful API на устройстве Cisco ASA, Nexus NX-API и Aristo eAPI (последние два – API на основе протокола HTTP, не поддерживающие REST). Также напомним, что каждый запрос NX-API и eAPI является HTTP-запросом POST, а URL один и тот же во всех запросах, в то время как полноценный RESTful API использует протокол HTTP как средство передачи (transport) с другим URL, основанным на ресурсе, указанном в запросе (например, ресурс интерфейса, ресурс маршрутизации или собственно сами маршруты).

Tlgm: @it_boooks

Ранее в текущей главе мы обращали внимание читателей на возможность использования специализированных библиотек в тех случаях, когда это разумно и оправданно. Здесь очень важно понимать, как правильно применять библиотеку requests, но для Arista существует библиотека с открытым исходным кодом pyeapi. Как и большинство библиотек, pyeapi предоставляет некоторый уровень абстракции для упрощения выполнения обобщенных задач. При использовании pyeapi можно не беспокоиться о правильном выполнении HTTP-запросов POST и о корректности структуры payload передаваемого объекта. Обо всем позаботится сама библиотека.
В приведенном ниже скрипте с помощью pyeapi выполняется один из тех же запросов, которые рассматривались в предыдущих примерах, несмотря на то что pyeapi поддерживает обмен данными только с одним устройством. Здесь можно видеть, что различие состоит в способе обмена данными с устройством на более низком уровне. После получения объекта ответа код обработки остается неизменным. Примечание: в этом примере скрипт использует файл конфигурации pyeapi. Создание корректного файла конфигурации eAPI выходит за рамки тематики данной книги.

```
from pyeapi import connect_to

if __name__ =="__main__":
    device = connect_to('eos-spine1')
    rsp = device.enable('show vlan brief')
    vlans = rsp[0]['result']['vlans']
    print('{:12}{:<10}'.format('VLAN ID', 'NAME'))
    for vlan_id, config in vlans.items():
        print('{:<12}{:<10}'.format(vlan_id,  config['name']))
```

Для получения более подробной информации о библиотеке pyeapi изучайте ее код и обратитесь к страницам документации.

Общее описание RESTCONF для IOS-XE

Для демонстрации основ работы с еще одним API на основе протокола HTTP, соблюдающим все принципы REST, рассмотрим более новый API для устройства Cisco IOS-XE. Этот API называется RESTCONF. RESTCONF API для IOS-XE представляет собой надежный RESTful API, использующий протокол HTTP для транспортировки, и поддерживает форматы кодировки данных JSON и XML. Поэтому разработчик сетевого ПО получает возможность выбрать наиболее предпочтительный для него формат данных.

RESTCONF (https://tools.ietf.org/html/draft-ietf-netconf-restconf-09) – это предварительный вариант (черновик) стандарта IETF, представляющий собой конкретную реализацию RESTful API. Это протокол на основе HTTP, который предоставляет программный интерфейс для доступа к данным, моделируемым на языке YANG.

Доступ к RESTCONF API осуществляется с помощью команды общей конфигурации `restconf` на устройстве IOS-XE 16.3.1 или более поздней версии. Отметим, что также необходимо выполнить конфигурирование с помощью команды общей конфигурации `ip http/s server`. ВАЖНО: команда `restconf` не поддерживается TAC до версии 16.6. В зависимости от версии эта команда может быть скрыта или, возможно, вообще не поддерживается TAC. Узнать об этом можно, только попытавшись выполнить команду. Также следует отметить, что некоторые вызовы API, использующие NETCONF/RESTCONF, изме-

Tlgm: @it_boooks

нились после версии 16.3. Инструкции по преобразованию вызовов API версии 16.3 (показанных в текущем разделе) в вызовы API версии 16.5+ можно найти на сайте GitHub (https://github.com/YangModels/yang/tree/master/vendor/cisco/xe/1651).

RESTCONF API поддерживает типы запросов GET, PUT, PATCH, POST и DELETE, что делает его чрезвычайно мощным инструментом выполнения сетевых операций, которые мы будем рассматривать в следующих подразделах.



Прежде чем приступить к изучению RESTful API для IOS-XE, необходимо отметить одно различие между использованием RESTCONF и обычного RESTful API. Это различие заключено в заголовках, формируемых в HTTP-запросе. В предыдущих примерах для Cisco NX-API, Cisco ASA RESTful API и Arista eAPI для заголовков `Content-Type` и `Accept` устанавливался тип `application/json`. Это обычный тип заголовков для API на основе HTTP, но поскольку отличительной особенностью RESTCONF является генерация вызовов API из моделей YANG, заголовки `Content-Type` и `Accept` могут быть различными. При работе с RESTCONF API можно пользоваться заголовками разнообразных типов. Для RESTCONF мы будем использовать преимущественно тип `application/vnd.yang.data+json`. Примечание: возможна замена `json` на `xml`.

Выполнение вызова API с использованием RESTCONF В первом примере будет выполнен вызов API, позволяющий получить полную рабочую конфигурацию, смоделированную в формате JSON.

```
Method: GET
URL: 'http://ios-csr1kv/restconf/api/config/native'
Accept-Type: application/vnd.yang.data+json
```

После выполнения этого вызова API в любом инструментальном приложении по вашему выбору будет выведен результат, показанный в следующем примере, но в действительности вы увидите полную конфигурацию, а ниже показан только ее фрагмент:

```
{
    "ned:native": {
        # вывод сокращен для экономии места
        "interface": {
            "GigabitEthernet": [
                {
                    "name": "1"
                },
                {
                    "name": "2"
                },
                {
                    "name": "3"
                },
                {
                    "name": "4"
                }
            ]
        },
        # вывод сокращен для экономии места
}
```

Tlgm: @it_boooks

Также можно добавить в URL параметр запроса для получения каждого элемента-потомка, являющегося частью иерархии JSON. Отметим, что в предыдущем примере для каждого интерфейса выведен только один ключ `name`. Параметр запроса `?deep` добавляет вывод полной конфигурации для каждого интерфейса, как показано в следующем примере.

```
Method: GET
URL: 'http://ios-csr1kv/restconf/api/config/native?deep'
Accept-Type: application/vnd.yang.data+json
```

```
{
    "ned:native": {
        # вывод сокращен для экономии места
        "interface": {
            "GigabitEthernet": [
                {
                    "negotiation": {
                        "auto": true
                    },
                    "ip": {
                        "access-group": {},
                        "arp": {
                            "inspection": {}
                        },,
                        "verify": {
                            "unicast": {}
                        },
                        "authentication": {},
                        "address": {
                            "primary": {
                                "mask": "255.255.255.0",
                                "address": "10.0.0.51"
                            }
                        },
                        "dhcp": {
                            "snooping": {},
                            "relay": {
                                "information": {}
                            }
                        },
                        "rsvp": {},
                        "ospf": {
                            "dead-interval": {},
                            "fast-reroute": {}
                        },
                        "igmp": {}
                    },
                    "standby": {},
                    "bandwidth": {},
                    "cdp": {},
                    "wrr-queue": {},
                    "bfd": {},
```

Tlgm: @it_boooks


```
                    "snmp": {
                        "trap": {
                            "link-status-capas": {
                                "link-status": {}
                            }
                        }
                    },
                    "power": {},
                    "logging": {},
                    "storm-control": {
                        "broadcast": {}
                    }
                    "encapsulation": {},
                    "name": "1",
                # вывод сокращен для экономии места
                },
                # выше показан вывод результата только для GigabitEthernet1
                # при использовании ?deep результаты повторяются для каждого интерфейса
}
```

Несмотря на то что и в этом примере бóльшая часть вывода пропущена, отметим, что при добавлении параметра `?deep` мы получаем все элементы-потомки конфигурации RESTCONF API на устройстве IOS-XE.

Вероятно, вы уже поняли, что приведенный ниже скрипт на языке Python предназначен для вывода ранее показанного результата с той же структурой, которая демонстрировалась во всех примерах применения библиотеки requests.

Этот скрипт используется для вывода обоих объектов JSON, показанных в двух последних примерах.

```
#!/usr/bin/env python

import json
import requests
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Accept': 'application/vnd.yang.data+json',
        'Content-Type': 'application/vnd.yang.data+json'
    }

    url = 'http://ios-csr1kv/restconf/api/config/native?deep'
    response = requests.get(url, headers=headers, auth=auth)

    response = json.loads(response.text)
    print(json.dumps(response, indent=4))
```

Более подробное исследование ответа RESTCONF API Теперь, когда понятно, как получить результат, аналогичный результату выполнения команды `show run`, с помощью вызова API, можно приступить к более детальному изучению последующих вызовов API, чтобы лучше понять структуру дерева, возвращаемую

Tlgm: @it_boooks

в объекте JSON. В предыдущих примерах основное внимание было уделено выводу конфигурации интерфейса.

Выполним запрос API для извлечения конфигураций IP-адресов, существующих на интерфейсе GigabitEthernet1. Упрощенная структура в формате JSON показана в следующем примере, чтобы специально выделить те ключи/значения JSON, которые представляют нужные нам данные интерфейса.

```
{
    "ned:native": {
        "interface": {
            "GigabitEthernet": [
                    "name": "1",
                    "ip": {
                        "address": {
                            "primary": {
                                "mask": "255.255.255.0",
                                "address": "10.0.0.51"
                            }
                        }
    # вывод сокращен для экономии места
```

Это те же выходные данные, которые можно использовать для тестирования и формирования последующих, более подробных HTTP-запросов GET. При конфигурировании адреса для GigabitEthernet1 можно заметить, что он представляет собой следующее путевое имя, формируемое на основе показанных выше выходных данных в формате JSON:

```
interface (dict) -> GigabitEthernet (list) -> ip (dict) -> address (dict)
```

Несмотря на то что значение `GigabitEthernet` является списком, мы используем значение ключа `name` в URL для вызова API, чтобы извлечь конкретный элемент. Рассмотрим эту операцию подробнее.

Для выполнения этого запроса нам опять необходим HTTP-запрос GET. Основное изменение вносится в URL, так как нужно получить доступ к другому ресурсу сетевого устройства.

Ниже показан URL, требуемый для выполнения API-запроса, рассматриваемого в нашем примере: http://ios-csr1kv/restconf/api/conig/native/interface/GigabitEthernet/1/ip/address.

После запуска обновленного скрипта мы получим вполне ожидаемый результат – только информацию об IP-адресе для GigabitEthernet1. Если бы у этого интерфейса были вторичные IP-адреса, они также были бы выведены.

```
cisco@cisco:~/netconf/xe$ python iosxe_restconf_GigE1.py
{
    "ned:address": {
        "primary": {
            "mask": "255.255.255.0",
            "address": "10.0.0.51"
        }
    }
}
```

Tlgm: @it_boooks

Использование HTTP-запросов PUT и PATCH при работе с RESTCONF на IOS-XE До сего момента в текущей главе мы выполняли запросы GET и POST с использованием Python-библиотеки requests для NX-OS, EOS, а теперь и для IOS-XE. Поскольку HTTP-запросы PUT и PATCH могут оказаться чрезвычайно полезными для управления сетевыми устройствами в будущем, необходимо рассмотреть возможности их применения при работе с IOS-XE RESTCONF API.

Сначала используем запрос PATCH для добавления сетевых команд OSPF в существующую конфигурацию OSPF.

Ниже показана текущая полная конфигурация OSPF на маршрутизаторе IOS-XE.

```
ios-csr1kv#show run | begin router ospf
router ospf 10
 router-id 100.100.100.100
 network 10.0.1.10 0.0.0.0 area 0
 network 10.0.2.100 0.0.0.0 area 0
!
```

Необходимо добавить следующие две сетевые команды определения кратчайших маршрутов в эту конфигурацию:

```
network 10.0.10.1 0.0.0.0 area 0
network 10.0.20.1 0.0.0.0 area 0
```

В первоначальной полной конфигурации обратите внимание на использование следующего URL для доступа к конфигурации OSPF: http://ios-csr1kv/restconf/api/conig/native/router.

В возвращаемом объекте конфигурации имеется ключ `ospf`. Внимательно изучая вывод данных предварительно сконфигурированного маршрутизатора, мы получаем возможность обнаружить объект OSPF, для которого требуется список словарей. Для каждого идентификатора ID процесса OSPF создается отдельный элемент списка.

В следующем примере показана краткая сводка данных JSON-представления конфигурации OSPF, извлеченного из вывода полной рабочей конфигурации:

```
"router": {
    "ospf": [
        {
            "id": 10,
            "router-id": "100.100.100.100",
            "network": [
                {
                    "ip": "10.0.1.10",
                    "mask": "0.0.0.0",
                    "area": 0
                },
                {
                    "ip": "10.0.2.100",
                    "mask": "0.0.0.0",
                    "area": 0
```

Tlgm: @it_boooks


```
                }
            ]
            # вывод сокращен для экономии места
        }
    ]
},
```

Напомним, что мы просто пытаемся добавить (то есть поставить «заплату» – patch) две дополнительные сетевые команды в существующую конфигурацию OSPF. Существуют различные способы формирования URL и тела запроса в зависимости от требуемого уровня иерархии данных о конфигурации. Чтобы продемонстрировать единообразие следующих двух примеров и свести к минимуму изменения в заголовках, воспользуемся следующим URL для добавления требуемых сетевых команд: http://ios-csr1kv/restconf/api/conig/native/router.

Для отправляемого объекта необходимо создать модель точно так же, как для данных, которые передавались в ответе на запрос GET в предыдущих примерах.



При использовании запроса PATCH необходимо включить в него только те пары ключ/значение, которые должны быть добавлены или обновлены.

Для добавления заданных маршрутов показанный ниже объект JSON передается на устройство с использованием URL: http://ios-csr1kv/restconf/api/conig/native/router:

```
{
    "ned:router": {
        "ospf": [
            {
                "id": 10,
                "network": [
                    {
                        "ip": "10.0.10.1",
                        "mask": "0.0.0.0",
                        "area": "0"
                    },
                    {
                        "ip": "10.0.20.1",
                        "mask": "0.0.0.0",
                        "area": "0"
                    }
                ]
            }
        ]
    }
}
```

Чтобы упростить скрипт на языке Python и сделать его более похожим на аналоги, используемые в реальной эксплуатационной среде, абстрагируемся от конкретной конфигурации OSPF (входные данные) и сохраним данные

Tlgm: @it_boooks

в YAML-файле. Показанный выше объект JSON можно легко смоделировать в файле формата YAML. Рассмотрим следующий пример:

```
---
ospf:
  - id: 10
    network:
      - ip: 10.0.10.1
        mask: 0.0.0.0
        area: 0
      - ip: 10.0.20.1
        mask: 0.0.0.0
        area: 0
```

Сохраним эту модель в файле *ospf-config.yml* и используем данный файл как источник входных данных для скрипта на языке Python. Примечание: созданный объект YAML равнозначен объекту, расположенному в ключе `ned:router`.

Применение языка Python для практического использования IOS-XE RESTCONF API Рассмотрим новый скрипт на языке Python, который добавит две требуемые сетевые команды в существующую конфигурацию OSPF.

```
#!/usr/bin/env python

import json
import requests
from requests.auth import HTTPBasicAuth
import yaml

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Accept': 'application/vnd.yang.data+json',
        'Content-Type': 'application/vnd.yang.data+json'
    }

    url = 'http://ios-csr1kv/restconf/api/config/native/router'

    ospf_config = yaml.load(open('ospf-config.yml').read())

    ospf_object_to_send = {
        "ned:router": ospf_config
    }

    response = requests.patch(url, data=json.dumps(ospf_object_to_send),
                              headers=headers, auth=auth)
    print(response.status_code)
```

При выполнении этого скрипта и проверке маршрутизатора можно видеть следующую новую конфигурацию OSPF:

```
router ospf 10
 router-id 100.100.100.100
 network 10.0.1.10 0.0.0.0 area 0
 network 10.0.2.100 0.0.0.0 area 0
```

Tlgm: @it_boooks


```
 network 10.0.10.1 0.0.0.0 area 0
 network 10.0.20.1 0.0.0.0 area 0
!
```



При необходимости можно вернуться к содержимому главы 4, чтобы вспомнить приемы работы с файлами средствами языка Python, а также к главе 5, где рассматривалась работа с объектами YAML.

Основы управления декларативной конфигурацией При планировании управления сетевыми конфигурациями из командной строки (CLI) добавление новых конфигураций является весьма простой задачей. При этом достаточно сложно удалять или отменять команды. Например, если требуется один экземпляр конфигурации OSPF на основе 50 сетевых команд, но из-за некоторых изменений в проекте потребуется всего лишь 2 сетевые команды, то задача усложняется. Необходимо точно знать, какие 48 команд должны быть отменены с помощью оператора `no`. Это трудоемкий и длительный процесс, поскольку работать приходится со всеми типами конфигураций на сетевом устройстве. Приведенный пример можно было бы упростить, применив совершенно другой подход – сосредоточение внимания на конфигурации, которая должна существовать на этом сетевом устройстве. Такая практическая методика применяется все чаще и успешнее благодаря новейшим API и прогрессивным способам мышления. Она получила название *декларативная конфигурация* (*declarative coniguration*).

Рассмотрим, как можно внести мельчайшее изменение в сформированную выше конфигурацию OSPF, а также скрипт для создания декларативной конфигурации OSPF при выполнении следующих команд.

```
router ospf 10
 router-id 10.10.10.10
 network 10.0.10.1 0.0.0.0 area 0
 network 10.0.20.1 0.0.0.0 area 0
!
```

При формировании декларативной конфигурации мы гарантируем, что не существует какой-либо другой конфигурации OSPF, кроме существующей в нашем единственном вызове API.

Изменения в выходных данных отображают новый идентификатор маршрутизатора, удаление двух изначально существовавших сетевых команд и добавление новых сетевых команд.

В первую очередь необходимо обновить YAML-файл и добавить ключ `router-id` с соответствующим значением `10.10.10.10`. Напомним, что содержимое этого YAML-файла становится единственной конфигурацией OSPF, существующей после вызова API.

```
---

ospf:
  - id: 10
    router-id: 10.10.10.10
```

Tlgm: @it_boooks


```
    network:
      - ip: 10.0.10.1
        mask: 0.0.0.0
        area: 0
      - ip: 10.0.20.1
        mask: 0.0.0.0
        area: 0
```

После обновления и сохранения YAML-файла требуется также внести изменение в сам вызов API, чтобы использовать HTTP-запрос PUT вместо PATCH.

```
response = requests.put(url, data=json.dumps(ospf_object_to_send),
                        headers=headers, auth=auth)
```

После внесения всех необходимых изменений мы просто запускаем скрипт и получаем требуемый результат – новую конфигурацию OSPF на маршрутизаторе:

```
router ospf 10
 router-id 10.10.10.10
 network 10.0.10.1 0.0.0.0 area 0
 network 10.0.20.1 0.0.0.0 area 0
!
```

При использовании RESTful API, предоставляющих такой тип управления и регулирования, необходима полная уверенность в том, что сформирован эффективный процесс внесения изменений. Если вы пытаетесь внести лишь незначительные изменения и дополнения и случайно отправляете запрос PUT, то последствия могут быть катастрофическими.

При осторожном и разумном освоении этого конкретного API вы можете начать с использования запросов PATCH и постепенно переходить к практике уверенного применения запросов PUT для декларативного управления отдельных разделов конфигурации.



Приведенный выше пример с использованием запроса PUT для декларативного конфигурирования всего содержимого ключа `router` характерен не только для конфигурации OSPF. Если вы выполняете вызовы этого API в своей рабочей среде, то с технической точки зрения существует возможность полного уничтожения всех конфигураций для других протоколов маршрутизации, что недопустимо. Тем не менее мы выбрали для демонстрации именно этот API, чтобы особенно выделить его мощь и потенциальную опасность при неправильном использовании из-за недостаточного понимания принципов его работы.

Другой, немного более безопасный вариант – декларативное управление только конкретным процессом с ID 10 для конфигурирования OSPF. Для этого можно воспользоваться следующим URL и объектом:

```
METHOD: PUT
URL: http://ios-csr1kv/restconf/api/config/native/router/ospf/10
BODY:
{
    "ned:ospf": [
```

Tlgm: @it_boooks


```
            {
                "id": 10,
                "network": [
                    {
                        "ip": "10.0.10.1",
                        "mask": "0.0.0.0",
                        "area": "0"
                    },
                    {
                        "ip": "10.0.20.1",
                        "mask": "0.0.0.0",
                        "area": "0"
                    }
                ]
            }
        ]
    }
```

Мы подробно рассмотрели основы автоматизации устройств, поддерживающих API на основе протокола HTTP, и теперь можно перейти к изучению аналогичного подхода с применением библиотеки ncclient на языке Python для автоматизации устройств, использующих NETCONF API.

Использование Python-библиотеки ncclient

Написанная на языке Python библиотека ncclient в действительности представляет собой NETCONF-клиента для среды Python. Это клиентское ПО, созданное для программирования процедур обмена данными с серверами NETCONF. Напомним, что в контексте текущей главы сервер NETCONF будет представлен сетевым устройством. Мы рассмотрим несколько примеров применения ncclient на устройствах Cisco IOS-XE, Cisco IOS-XR и Juniper Junos.



Для установки библиотеки ncclient воспользуйтесь утилитой `pip`:

```
pip install ncclient
```

После установки библиотеки ncclient можно сразу начать выполнение вызовов NETCONF API на сетевых устройствах. Изучать функциональные возможности библиотеки удобнее в интерактивном интерпретаторе Python.

После входа в интерпретатор в первую очередь необходимо импортировать модуль `manager` из пакета ncclient.

```
>>> from ncclient import manager
>>>
```

Основная функция из модуля `manager`, которую мы будем интенсивно использовать, отвечает за установление постоянного соединения с устройством. Следует всегда помнить, что поскольку протокол NETCONF работает поверх протокола SSH, необходимо постоянное соединение с сохранением состояния (в отличие от рассмотренных в предыдущих разделах RESTful API, которые не

Tlgm: @it_boooks

поддерживают сохранение состояния). Это функция connect(), принимающая несколько параметров, таких как имя хоста или IP-адрес, номер порта и регистрационные данные. В приведенном ниже примере вы увидите и другие параметры, которые будут оставаться неизменными в последующих примерах. Эти параметры определяют конфигурацию и свойства SSH-соединения более низкого уровня.

```
>>> device = manager.connect(host='ios-csr1kv', port=830, username='ntc',
...                          password='ntc123', hostkey_verify=False,
...                          device_params={}, allow_agent=False,
...                          look_for_keys=False)
...
>>>
```

Сразу после вызова функции connect() создается сеанс NETCONF с сетевым устройством и возвращается объект типа ncclient.manager.Manager. Его можно увидеть, если вывести на экран содержимое переменной, в которой сохранен этот объект.

```
>>> print(device)
<ncclient.manager.Manager object at 0x7f0420059d90>
>>>
```

В приведенном примере device – это экземпляр объекта Manager из библиотеки ncclient. Исследуем данный объект более подробно в интерактивном интерпретаторе Python.

Исследование объекта Manager

В первую очередь необходимо рассмотреть вывод функции dir() для переменной device и воспользоваться функцией help() для получения информации о некоторых выведенных объектах.

```
>>> dir(device)
['_Manager__set_async_mode', '_Manager__set_raise_mode', '_Manager__set_timeout',
'...methods removed for brevity... '_device_handler', '_raise_mode', '_session',
'_timeout', 'async_mode', 'channel_id', 'channel_name', 'client_capabilities',
'close_session', 'commit', 'connected', 'copy_config', 'delete_config',
'discard_changes', 'dispatch', 'edit_config', 'execute', 'get', 'get_config',
'get_schema', 'kill_session', 'lock', 'locked', 'poweroff_machine',
'raise_mode', 'reboot_machine', 'scp', 'server_capabilities', 'session',
'session_id', 'timeout', 'unlock', 'validate']
>>>
```

Некоторые методы, показанные в выводе функции dir(), предназначены для установки и тонкой настройки соединения, но нас интересуют методы, предназначенные непосредственно для выполнения операций NETCONF.

При формировании XML-документов и передаче их в интерактивном сеансе NETCONF мы видели, что внутри тега <rpc> специальный XML-тег определяет требуемую операцию. Ранее мы рассматривали две операции: <get> и <edit-config>. В выводе предыдущего примера также можно видеть два ме-

Tlgm: @it_boooks

тода с именами get и edit-config. Кроме того, в выводе перечисляются методы, соответствующие другим стандартным операциям NETCONF, которые мы рассматривали ранее: copy-config, get-schema, lock, unlock, validate и прочие.

Метод get Теперь нам известно, что методы объекта device соответствуют операциям NETCONF. Возможности их практического использования можно узнать с помощью встроенной функции help().

```
>>>
>>> help(device.get)

Help on method wrapper in module ncclient.manager:

wrapper(self, *args, **kwds) method of ncclient.manager.Manager instance
    Retrieve running configuration and device state information.

    *filter* specifies the portion of the configuration to
             retrieve (by default entire configuration is retrieved)

    :seealso: :ref:`filter_params`
(END)
```

Текст, выводимый функцией help(), автоматически извлечен из блока docstring, содержащегося непосредственно в исходном коде. Таким образом, действительную помощь можно получить только в том случае, если автор ncclient позаботился о написании соответствующего блока docstring. Если вспомогательная информация отсутствует, то вам не повезло (таковы особенности проектов с открытым исходным кодом), тем не менее попробуйте обратиться с запросом на GitHub.

В информации об использовании метода get() указано, что он принимает переменное количество параметров и ключевых слов-аргументов, обозначаемых символами * и **. Необязательное ключевое слово, которое можно использовать, обозначено именем filter. Некоторые устройства поддерживают возврат полных конфигураций, но в следующих примерах все внимание будет сосредоточено на применении параметра filter для выборочного получения частей конфигурации в виде данных, закодированных в формате XML.

Извлечение конфигураций устройства Cisco IOS-XE с помощью библиотеки ncclient

Ранее в текущей главе при изучении использования NETCONF мы пользовались следующим XML-документом как способом выполнения запроса на маршрутизаторе Cisco IOS-XE с целью получения конфигурации для интерфейса GigabitEthernet1:

```
<?xml version="1.0"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
      <filter type="subtree">
        <native xmlns="http://cisco.com/ns/yang/ned/ios">
         <interface>
          <GigabitEthernet>
```

Tlgm: @it_boooks


```
          <name>1</name>
        </GigabitEthernet>
      </interface>
    </native>
  </filter>
</get>
</rpc>
]]>]]>
```



Выше в текущей главе было отмечено, что в книге используется IOS-XE версии 16.3.1. Модели устройств изменились по сравнению с версией 16.3 в версиях 16.5 и 16.6. Инструкции по преобразованию вызовов API версии 16.3 (показанных в этом разделе) в вызовы версий 16.5+ можно найти на GitHub (https://github.com/YangModels/yang/tree/master/vendor/cisco/xe/1651).

Это был вызов удаленной процедуры NETCONF RPC с использованием операции `<get>` для передачи фильтра на устройство. Если требуется сделать это средствами языка Python, то необходимо создать точно такой же объект фильтра. Такой объект формируется в виде XML-строки.

```
>>> get_filter = """
...     <native xmlns="http://cisco.com/ns/yang/ned/ios">
...       <interface>
...         <GigabitEthernet>
...           <name>1</name>
...         </GigabitEthernet>
...       </interface>
...     </native>
... """
>>>
```



Напомним, что три символа двойных кавычек в Python обозначают многострочный комментарий и используются для создания очень длинной фразы, размещенной на нескольких строках, которая может применяться как значение переменной.

После создания фильтра он передается как параметр в метод `get()`.

```
>>> nc_get_reply = device.get(('subtree', get_filter))
>>>
```



В этой книге используются только фильтры типа subtree. NETCONF и ncclient также поддерживают фильтры xpath, основанные на специальных функциональных возможностях NETCONF, которые обязательно должно поддерживать сетевое устройство.

В метод `get()` непременно должен передаваться только один объект фильтра – кортеж из двух элементов: типа фильтра и самого фильтра/выражения.

В приводимых здесь примерах в качестве фильтров используются строки в формате XML, но возможно также использование собственных объектов XML (объектов etree). Объекты-строки применяются потому, что они более удобны для чтения человеком и с ними гораздо проще начинать освоение технологии. Возможно, вы предпочтете объекты etree, если необходимо динамически формировать объект фильтра. В следующих примерах мы продемонстрируем применение элементов etree в качестве объектов фильтров.

Tlgm: @it_boooks

Итак, мы выполнили запрос NETCONF к устройству и сохранили результат в переменной `nc_get_reply`.

Исследование ответа NETCONF с помощью ncclient

Выведем содержимое объекта ответа, то есть переменной `nc_get_reply`.

```
>>> print(nc_get_reply)
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:e103ecdf-9713-46c0-8769-0e574d9b4489"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"><data>
<native xmlns="http://cisco.com/ns/yang/ned/ios"><interface>
<GigabitEthernet><name>1</name><negotiation><auto>true</auto>
</negotiation><vrf><forwarding>MANAGEMENT</forwarding></vrf>
<ip><address><primary><address>10.0.0.51</address><mask>255.255.255.0</mask>
</primary></address></ip></GigabitEthernet></interface></native>
</data></rpc-reply>
>>>
```

Также можно узнать тип данных полученного объекта.

```
>>> type(nc_get_reply)
<class 'ncclient.operations.retrieve.GetReply'>
>>>
```

Можно просматривать ответ в формате XML в виде простого вывода на экран, но поскольку это объект библиотеки ncclient типа `GetReply`, а не встроенный объект языка Python, такой как строка, словарь или список, необходимо будет более подробно изучить атрибуты, присущие этому объекту.

Объект `GetReply` обладает несколькими встроенными атрибутами, которые упрощают получение ответов при работе с NETCONF. Рассмотрим некоторые атрибуты более подробно.

Атрибуты `data` и `data_ele` возвращают сам этот объект, то есть это ответ, представленный как обычный объект XML. Для проверки типов таких объектов можно воспользоваться функцией `type()`.

```
>>> type(nc_get_reply.data)
<type 'lxml.etree._Element'>
>>>
>>> type(nc_get_reply.data_ele)
<type 'lxml.etree._Element'>
>>>
```

Таким образом, при обращении к обычным XML-объектам фактически происходит обращение к объектам типа `lxml.etree._Element`.

```
>>> print(nc_get_reply.data_ele)
<Element {urn:ietf:params:xml:ns:netconf:base:1.0}data at 0x7f041acbab48>
>>>
>>> print(nc_get_reply.data)
<Element {urn:ietf:params:xml:ns:netconf:base:1.0}data at 0x7f041acbab48>
>>>
```

Tlgm: @it_boooks

Отметим, что в обоих случаях выводится один и тот же объект. Тип этого объекта был определен выше. Такой объект может использоваться как фильтр вместо XML-строки. В этом разделе мы будем использовать атрибут `data`, хотя можно пользоваться и атрибутом `data_ele`, поскольку оба атрибута представляют один и тот же объект.

Для преобразования обычного XML-объекта в строку можно воспользоваться Python-библиотекой `lxml`, точнее функцией этой библиотеки `tostring()`. Напомним, что библиотека `lxml` рассматривалась в главе 5.

```
>>> from lxml import etree
>>>
>>> as_string = etree.tostring(nc_get_reply.data)
>>>
>>> print(as_string)
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
<native xmlns="http://cisco.com/ns/yang/ned/ios"><interface>
<GigabitEthernet><name>1</name><negotiation><auto>true</auto>
</negotiation><vrf><forwarding>MANAGEMENT</forwarding></vrf>
<ip><address><primary><address>10.0.0.51</address>
<mask>255.255.255.0</mask></primary></address></ip>
</GigabitEthernet></interface></native></data>
>>>
```

Отметим, что вывод не отформатирован, но прочесть его можно. Если воспользоваться необязательным параметром `pretty_print` и установить для него значение `True`, то ответ станет гораздо более удобным для чтения.

```
>>> as_string = etree.tostring(nc_get_reply.data, pretty_print=True)
>>>
>>> print(as_string)
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/ned/ios">
    <interface>
      <GigabitEthernet>
        <name>1</name>
        <negotiation>
          <auto>true</auto>
        </negotiation>
        <vrf>
          <forwarding>MANAGEMENT</forwarding>
        </vrf>
        <ip>
          <address>
            <primary>
              <address>10.0.0.51</address>
              <mask>255.255.255.0</mask>
            </primary>
          </address>
        </ip>
```

Tlgm: @it_boooks


```
      </GigabitEthernet>
    </interface>
  </native>
</data>

>>>
```

Исследование других атрибутов ответа, полученного с помощью библиотеки ncclient

После подробного исследования атрибутов `data` и `data_ele` рассмотрим другой атрибут с именем `xml`. Атрибут `xml` возвращает ответ в виде XML-строки. Это можно проверить с помощью функции `type()`.

```
>>> type(nc_get_reply.xml)
<type 'str'>
>>>
```

Содержимое атрибута можно вывести как обычную строку в формате XML.

```
>>> print(nc_get_reply.xml)
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:e103ecdf-9713-46c0-8769-0e574d9b4489"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"><data>
<native xmlns="http:///cisco.com/ns/yang/ned/ios"><interface>
<GigabitEthernet><name>1</name><negotiation><auto>true</auto>
</negotiation><vrf><forwarding>MANAGEMENT</forwarding></vrf><ip>
<address><primary><address>10.0.0.51</address><mask>255.255.255.0</mask>
</primary></address></ip></GigabitEthernet></interface>
</native></data></rpc-reply>
>>>
```

При изучении вывода в формате XML из атрибута `xml` обратите внимание на то, что XML-тегом самого верхнего уровня является `<rpc-reply>`, а все данные ответа размещены внутри XML-тега `<data>`.

Точно так же, как XML-объект можно преобразовать в строку с помощью функции `etree.tostring()`, возможно и обратное преобразование строки в XML-объект с помощью функции `etree.fromstring()`.

```
>>> as_object = etree.fromstring(nc_get_reply.xml)
>>>
>>> print(as_object)
<Element {urn:ietf:params:xml:ns:netconf:base:1.0}rpc-reply at 0x7fc33365f9e0>
>>>
```

О пространстве имен XML можно не беспокоиться, так как очевидно, что в данном примере именем XML-объекта является `rpc-reply`. При выводе XML-объекта его имя всегда представлено XML-тегом самого верхнего уровня, то есть в нашем случае `rpc-reply`.

До сих пор использовалась операция NETCONF `<get>`, реализованная через метод `get()` объекта устройства ncclient, но пока еще не было показано, как вы-

Tlgm: @it_boooks

полнить синтаксический разбор (парсинг) и извлечь информацию из сообщения, содержащего ответ RPC в формате XML.

Мы видели, что в ответе содержится IP-адрес и маска, сконфигурированные для интерфейса GigabitEthernet1. Оба элемента являются элементами-потомками XML-объекта `<address>`. Рассмотрим этот объект более подробно:

```
<address>
  <primary>
    <address>10.0.0.51</address>
    <mask>255.255.255.0</mask>
  </primary>
</address>
```

Это означает, что возможно существование более одного объекта `<address>` с первичной (primary) и вторичной (secondary) конфигурациями адресов.

Извлечем первичный IP-адрес и маску и сохраним их в отдельных переменных. Это будет сделано в два этапа. На первом этапе будет извлечен объект `<primary>`, на втором – элементы `<address>` и `<mask>`.

```
>>> primary = nc_get_reply.data.find(
    './/{http://cisco.com/ns/yang/ned/ios}primary')
>>>
```

В этом примере впервые используется метод `find()` для объектов типа `etree._Element`. Метод `find()` предоставляет простой способ поиска полного XML-объекта по заданному XML-тегу с использованием специального выражения, обозначенного префиксом `.//`. Так как требуется извлечь объект `<primary>` и все его элементы-потомки, можно было бы попытаться сразу выполнить следующий пример, но в действительности он не сработает:

```
>>> primary = nc_get_reply.data.find('.//primary')
>>>
```

Эта команда пытается извлечь XML-элемент с тегом `<primary>`. Здесь необходимо предупредить о том, что при использовании пространств имен XML действительное имя тега сформировано из наименования пространства имен, объединенного с именем тега, то есть *{namespace}tag*. Другой вариант, если используется псевдоним (alias) пространства имен XML: *alias*:_tag_. В нашем случае псевдоним не применяется, поэтому полное имя объекта `<primary>` обозначается как `{http://cisco.com/ns/yang/ned/ios}primary`.

В действительности в приведенном выше примере существуют два пространства имен. Попробуем разобраться, какое из них реально используется.

```
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/ned/ios">
```

Можно последовательно выводить по одному объекту-потомку, чтобы увидеть, какое пространство имен используется на самом деле. В приведенном примере по умолчанию определено пространство имен `urn:ietf:params:xml:ns:netconf:base:1.0`, и при выводе

Tlgm: @it_boooks

отдельного объекта мы видим только его. Следующее пространство имен в иерархии замещается пространством имен по умолчанию для всех потомков элемента <native>.

Чтобы лучше понять эту особенность, выведем элемент <primary> как строку.

```
>>> print(etree.tostring(primary, pretty_print=True))
<primary xmlns="http://cisco.com/ns/yang/ned/ios"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <address>10.0.0.51</address>
  <mask>255.255.255.0</mask>
</primary>
>>>
```

Отметим, что это XML-строка, начинающаяся с тега <primary>. Теперь можно извлечь из нее интересующие нас значения – IP-адрес и маску подсети. Применим тот же подход с использованием метода find() для созданного объекта primary. На этот раз продвинемся немного глубже по иерархии, обращаясь к атрибуту text полученного XML-объекта. Именно в этом атрибуте содержатся требуемые значения, то есть IP-адрес и маска подсети.

```
>>> ipaddr = primary.find('.//{http://cisco.com/ns/yang/ned/ios}address')
>>> ipaddr.text
'10.0.0.51'
>>>
>>> mask = primary.find('.//{http://cisco.com/ns/yang/ned/ios}mask')
>>> mask.text
'255.255.255.0'
>>>
```



Возможно, вы подумали: «Извлечение значений на основе пространств имен – это очень трудоемкая задача», и вы абсолютно правы. Тем не менее необходимо помнить о следующем:

- вам всегда известно пространство имен из создаваемого объекта запроса. Необходимо просто объединить две строки;
- существует возможность создать функцию для удаления обозначений пространств имен из XML-объекта перед выполнением синтаксического разбора (парсинга) формата XML для существенного упрощения процесса.

В примерах этого раздела было продемонстрировано извлечение отдельных значений, таких как первичный IP-адрес и маска подсети, но что делать, если необходимо извлечь все объекты с многочисленными значениями? В следующем примере мы будем работать с интерфейсом, для которого определено несколько вторичных IP-адресов.

Добавление в запрос фильтра для минимизации объема данных ответа

В предыдущих примерах несколько вторичных IP-адресов вручную добавлялись для интерфейса GigabitEthernet4, но теперь те же самые операции будут выполняться для того, чтобы увидеть, как выглядит объект ответа после NETCONF-операции <get> только для одного интерфейса GigabitEthernet4.

Tlgm: @it_boooks


```
>>> get_filter = """
...     <native xmlns="http://cisco.com/ns/yang/ned/ios">
...      <interface>
...       <GigabitEthernet>
...        <name>4</name>
...       </GigabitEthernet>
...      </interface>
...     </native>
... """
>>>
>>> nc_get_reply = device.get(('subtree', get_filter))
>>>
>>> print(etree.tostring(nc_get_reply.data, pretty_print=True))
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/ned/ios">
    <interface>
      <GigabitEthernet>
        <name>4</name>
        <negotiation>
          <auto>true</auto>
        </negotiation>
        <ip>
          <address>
            <primary>
              <address>10.4.4.1</address>
              <mask>255.255.255.0</mask>
            </primary>
            <secondary>
              <address>20.2.2.1</address>
              <mask>255.255.255.0</mask>
              <secondary/>
            </secondary>
            <secondary>
              <address>22.2.2.1</address>
              <mask>255.255.255.0</mask>
              <secondary/>
            </secondary>
            <secondary>
              <address>24.2.2.1</address>
              <mask>255.255.255.0</mask>
              <secondary/>
            </secondary>
          </address>
        </ip>
      </GigabitEthernet>
    </interface>
  </native>
</data>

>>>
```

Tlgm: @it_boooks

В этом примере наша главная цель – вывести на экран все вторичные IP-адреса и соответствующие им маски подсетей.

Один из вариантов решения – использовать цикл `for` и метод `iter()` объекта `etree`.

В следующем примере демонстрируется применение пространств имен при создании переменной `xmlns`, которую можно использовать для формирования шаблона строки с помощью метода `format()`.

```
>>> xmlns = '{http://cisco.com/ns/yang/ned/ios}'
>>> address_container = nc_get_reply.data.find('.//{}address'.format(xmlns))
>>> for secondary in address_container.iter('{}secondary'.format(xmlns)):
...     if secondary:
...         print(secondary.find('.//{}address'.format(xmlns)).text)
...         print(secondary.find('.//{}mask'.format(xmlns)).text)
...         print('-' * 10)
...
20.2.2.1
255.255.255.0
----------
22.2.2.1
255.255.255.0
----------
24.2.2.1
255.255.255.0
----------
>>>
```

Здесь показан только один способ извлечения вторичных адресов (или любых объектов, представленных несколькими экземплярами). Рассмотрим более простой подход. Поскольку у нас уже имеется `address_container`, содержащий все первичные и вторичные адреса, воспользуемся им для вывода каждого адреса, но сначала извлечем все элементы `address`, используя для этого метод `findall()` объекта `etree._Element`.

```
>>> all_addresses = address_container.findall('.//{}address'.format(xmlns))
>>>
>>> for item in all_addresses:
...     print(item.text)
...
10.4.4.1
20.2.2.1
22.2.2.1
24.2.2.1
>>>
```

Метод `findall()` подтверждает свою полезность в тех случаях, когда необходимо извлечь несколько элементов одного и того же типа.

Теперь вы знаете почти все о выполнении запросов NETCONF `<get>`. Тем не менее рассмотрим еще один пример, но на этот раз мы будем работать с маршрутизатором Juniper Junos vMX.

Tlgm: @it_boooks

Получение конфигураций устройства Juniper vMX Junos с помощью библиотеки ncclient

В текущий момент на устройстве Juniper vMX имеются две строки пароля сообщества (community string) конфигурации SNMP, защищенные от записи (read-only). Для проверки ниже приведен результат выполнения команды `show snmp` в режиме конфигурирования:

```
ntc@vmx1# show snmp
community public {
    authorization read-only;
}
community networktocode {
    authorization read-only;
}
[edit]
ntc@vmx1#
```

Необходимо извлечь имя каждой строки пароля сообщества и уровень авторизации для каждой строки.

Juniper обеспечивает функциональность в режиме интерфейса командной строки (CLI), поэтому можно получить ожидаемый ответ в формате XML, а таже отформатировать его, применив в конвейере команду `display xml`.

```
ntc@vmx1# show snmp | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1F4/junos">
    <configuration junos:changed-seconds="1482848933"
junos:changed-localtime="2016-12-27 14:28:53 UTC">
            <snmp>
                <community>
                    <name>public</name>
                    <authorization>read-only</authorization>
                </community>
                <community>
                    <name>networktocode</name>
                    <authorization>read-only</authorization>
                </community>
            </snmp>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
[edit]
ntc@vmx1#
```

Теперь, когда известно, что именно возвращает этот запрос NETCONF, написать соответствующий код Python гораздо проще. В любом ответе на NETCONF-запрос `get` к устройству Junos `<configuration>` непременно должен быть XML-тегом самого верхнего уровня при сборе информации о состоянии конфигурации. Внутри этого элемента можно сформировать требуемый

Tlgm: @it_boooks

фильтр, подобрав его компоненты из XML-текста, выводимого в интерфейсе командной строки. Пример такого фильтра для запроса конфигурации SNMP может выглядеть следующим образом:

```
get_filter = """
<configuration>
  <snmp>
  </snmp>
</configuration>
"""
```

Кроме того, создается экземпляр нового объекта устройства, и используется имя переменной `vmx` для установления соединения с маршрутизатором Juniper vMX.

```
>>> vmx = manager.connect(host='junos-vmx', port=830, username='ntc',
...                       password='ntc123', hostkey_verify=False,
...                       device_params={}, allow_agent=False,
...                       look_for_keys=False)
...
>>>
```

Далее создается запрос точно таким же способом, как это делалось ранее. После создания запроса проверяется вывод результата, и в терминале выводится XML-строка при помощи атрибута ответа `xml`.

```
>>> nc_get_reply = vmx.get(('subtree', get_filter))
>>>
>>> print(nc_get_reply.xml)
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1F4/junos"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:29121999-68b8-4dc5-9374-63bab673677b">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
junos:changed-seconds="1482848933"
junos:changed-localtime="2016-12-27 14:28:53 UTC">
    <snmp>
        <community>
            <name>public</name>
            <authorization>read-only</authorization>
        </community>
        <community>
            <name>networktocode</name>
            <authorization>read-only</authorization>
        </community>
    </snmp>
</configuration>
<database-status-information>
<database-status>
<user>ntc</user>
<terminal>p0</terminal>
```

Tlgm: @it_boooks


```
<pid>37551</pid>
<start-time junos:seconds="1483116118">2016-12-30 16:41:58 UTC</start-time>
<idle-time junos:seconds="684">00:11:24</idle-time>
<edit-path>[edit]</edit-path>
</database-status>
<database-status>
<user>ntc</user>
<terminal></terminal>
<pid>37643</pid>
<start-time junos:seconds="1483117954">2016-12-30 17:12:34 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
>>>
```



В ответе Juniper содержатся еще и метаданные о запросе, которые невозможно увидеть в интерфейсе командной строки: например, имя пользователя, выполнившего запрос, и время начала выполнения запроса.

Как уже было отмечено ранее, главной задачей является синтаксический анализ (парсинг) ответа с сохранением строки пароля сообщества и типа авторизации для каждого сообщества. Вместо простого вывода в терминале сохраним эти данные в виде списка словарей Python.

Для этого потребуется выполнение некоторых этапов, описанных выше. Единственное различие состоит в том, что данные сохраняются, а не выводятся на экран. Напомним, что необходимо либо удалить пространства имен XML с помощью специально написанного для этого кода, либо определить пространство имен при выводе объекта ответа в виде строки XML.

```
>>> snmp_list = []
>>>
>>> xmlns = '{http://xml.juniper.net/xnm/1.1/xnm}'
>>>
>>> communities = nc_get_reply.data.findall('.//{}community'.format(xmlns))
>>>
>>> for community in communities:
...     temp = {}
...     temp['name'] = community.find('.//{}name'.format(xmlns)).text
...     temp['auth'] = community.find('.//{}authorization'.format(xmlns)).text
...     snmp_list.append(temp)
...
>>>
>>> print(snmp_list)
[{'name': 'public', 'auth': 'read-only'}, {'name': 'networktocode',
                                           'auth': 'read-only'}]
>>>
```

В приведенном примере показано, как выполнить запросы NETCONF для получения данных конфигурации, но в следующем разделе мы рассмотрим

Tlgm: @it_boooks

способы внесения изменений в конфигурации с помощью NETCONF API, используя для этого операцию <edit-config>.

Изменение конфигурации Cisco IOS-XE с помощью библиотеки ncclient

Операция NETCONF <edit-config> напрямую связана с методом edit_config() объекта целевого устройства, предоставляемого библиотекой ncclient. Ранее уже был продемонстрирован пример практического применения операции <edit-config> при изучении использования NETCONF API. При внесении изменений в конфигурацию необходимо определить два элемента. Первый элемент <target> определяет хранилище данных конфигурации, содержимое которого будет изменено в выполняемом запросе. Можно выбрать один из трех вариантов хранилища: running (текущее рабочее), startup (начальное) или candidate (кандидат). Второй параметр <config> должен быть строкой в формате XML или объектом XML, который определяет требуемые изменения в конфигурации.

В первом примере будет сконфигурирована новая строка пароля сообщества SNMP с использованием YANG-модели network element driver (ned), которая уже использовалась ранее. Эта модель определена пространством имен http://cisco.com/ns/yang/ned/ios. Применяя эту конкретную модель, можно воспользоваться одним фильтром для возврата XML-объекта, описывающего иерархию, необходимую для выполнения последующих вызовов API. Выше уже был продемонстрирован такой подход с использованием RESTful API на IOS-XE. Для RESTful API был определен URL http://ios-csr1kv/restconf/api/conig/native/. Тот же самый фильтр запроса get для NETCONFвыглядит следующим образом:

```
get_filter = """
    <native xmlns="http://cisco.com/ns/yang/ned/ios">
    </native>
"""
```



Если устройство поддерживает и NETCONF и RESTCONF, то можно воспользоваться такими инструментальными средствами, как Postman, для взаимодействия с RESTful HTTP API или применить кодировку XML для более удобного изучения объектов, требуемых для NETCONF.

После передачи показанного выше фильтра устройство возвращает почти полную конфигурацию. Здесь вывод результата не показан, потому что он занимает слишком много места, но если вы самостоятельно выполните запрос с указанным выше фильтром, то обратите внимание на <snmp-server>, являющийся элементом-потомком тега <data>.

Это означает, что можно добавить элемент <snmp-server> в фильтр, чтобы выборочно извлечь только конфигурацию SNMP.

```
get_filter = """
    <native xmlns="http://cisco.com/ns/yang/ned/ios">
```

Tlgm: @it_boooks


```
      <snmp-server>
      </snmp-server
    </native>
"""
```

Несмотря на то что основной целью является внесение изменений в конфигурацию, здесь выполняется запрос get, чтобы увидеть, как нужно структурировать данные при передаче их на устройство в запросе <edit-config>.

```
>>> nc_get_reply = device.get(('subtree', get_filter))
>>>
>>> print(etree.tostring(nc_get_reply.data, pretty_print=True))
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <native xmlns="http://cisco.com/ns/yang/ned/ios">
    <snmp-server>
      <community>
        <name>networktocode</name>
        <RO/>
      </community>
      <community>
        <name>private</name>
        <RW/>
      </community>
      <community>
        <name>public</name>
        <RO/>
      </community>
    </snmp-server>
  </native>
</data>

>>>
```

Опередилив, что моделью данных SNMP является XML, можно сформировать новый XML-объект конфигурации. Добавим новую строку пароля сообщества secure, в которой определены права на чтение и запись (read-write). Соответствующая модель выглядит следующим образом:

```
  <native xmlns="http://cisco.com/ns/yang/ned/ios">
    <snmp-server>
      <community>
        <name>secure</name>
        <RW/>
      </community>
    </snmp-server>
  </native>
```

Последним тегом, который включает в себя весь сформированный документ, является <config>. Этот тег часто требуется при выполнении операции <edit-config>, как было отмечено при изучении стека протоколов NETCONF в одном из предыдущих разделов текущей главы.

Tlgm: @it_boooks

Ниже приведена новая переменная, представляющая требуемый объект конфигурации.

```
config_filter = """
    <config>
      <native xmlns="http://cisco.com/ns/yang/ned/ios">
        <snmp-server>
          <community>
            <name>secure</name>
            <RW/>
          </community>
        </snmp-server>
      </native>
    </config>
"""
```

В настоящее время IOS-XE не поддерживает конфигурацию-кандидата, поэтому указанное изменение вносится в хранилище данных текущей рабочей конфигурации. Это делается с помощью метода `edit_config()`, в который передаются два параметра: `target` и `config`.

```
>>> response = device.edit_config(target='running', config=config_filter)
>>>
```

Если проверить новую конфигурацию с использованием интерфейса командной строки или с помощью запроса `get`, то можно увидеть, что на устройстве теперь определены четыре строки пароля сообщества, в том числе строка `secure`.

Выполнение NETCONF-операции delete с помощью библиотеки ncclient В предыдущем разделе было продемонстрировано внесение изменений в конфигурацию устройства с помощью операции `<edit-config>`. Ранее было отмечено и показано в табл. 7.3, что по умолчанию выбирается тип операции `merge`. Тип операции можно изменить, используя для этого XML-атрибут `operation`.

В предыдущем примере была добавлена строка пароля сообщества SNMP `secure` с помощью операции `merge` (выбранной по умолчанию). В следующем примере показано, как удалить эту строку с помощью операции `delete`.

Для этого необходимо изменить только XML-строку (или объект) конфигурации. Обратите внимание на новую строку в формируемой переменной: `operation="delete"`.

```
config_filter = """
    <config>
      <native xmlns="http://cisco.com/ns/yang/ned/ios">
        <snmp-server>
          <community operation="delete">
            <name>secure</name>
            <RW/>
          </community>
        </snmp-server>
```

Tlgm: @it_boooks


```
      </native>
    </config>
"""
```

После выполнения этого запроса с помощью того же метода `edit_config()` строка пароля сообщества `secure` будет удалена.

Теперь вы знаете, как добавлять данные в конфигурацию с помощью операции `operation=merge` и как удалять данные из конфигурации с использованием операции `operation=delete`. Но как можно обеспечить существование только одного сообщества `secure`?

Выполнение NETCONF-операции replace с помощью библиотеки ncclient Существует несколько способов замены заданного фрагмента иерархии конфигурации в формате XML. Можно выполнить вызов API для конфигурирования всех требуемых строк пароля сообщества SNMP, затем извлечь все текущие сконфигурированные сообщества SNMP в другом вызове API, обработать полученный ответ в цикле и выполнить операцию удаления всех сообществ, которые не нужны. Это не самый худший способ, но NETCONF предлагает более эффективное решение, и вам оно, разумеется, уже известно.

При работе с NETCONF можно воспользоваться операцией `replace` вместо `merge` или `delete`. Эта операция аналогична запросу PUT, поддерживаемому RESTCONF API.

Выполним обновление XML-строки конфигурации, добавив в нее `operation=replace`, и посмотрим, к какому результату приведет это изменение.

```
config_filter = """
    <config>
      <native xmlns="http://cisco.com/ns/yang/ned/ios">
        <snmp-server operation="replace">
          <community>
            <name>secure</name>
            <RW/>
          </community>
        </snmp-server>
      </native>
    </config>
"""
```

Выполняется тот же запрос `<edit-config>`, который теперь позволяет обеспечить существование в конфигурации `<snmp-server>` единственной строки пароля сообщества `secure`. Обратите особое внимание на то, что строка `operation="replace"` располагается в другом месте XML-иерархии, по сравнению с предыдущим примером. Размещение этой строки как атрибута тега `<snmp-server>` позволяет полностью заменить конфигурацию SNMP. Если выполнить вызов API с атрибутом `operation="delete"` в той же строке, то будут удалены все соответствующие конфигурации SNMP. По аналогии с RESTCONF-запросом PUT то же самое справедливо для NETCONF-операций `replace`, поэтому будьте внимательны и осторожны при их использовании.

Tlgm: @it_boooks



Эксперименты с операциями `delete` и `replace` требуют исключительного внимания и чрезвычайной осторожности. Операция `merge` устанавливается по умолчанию вполне обоснованно, потому что с ее помощью можно лишь добавить или обновить данные конфигурации. Очень важно помнить, что если неправильно разместить строку `operation="delete"` в XML-иерархии, то результат выполнения операции удаления может оказаться катастрофическим для всей сети. Поэтому необходимо всегда предварительно тестировать запросы в лабораторных условиях или в специально предназначенных для тестов средах – «песочницах». Никогда не проводите тестирование в реальной эксплуатационной среде.

Внесение изменений в конфигурацию Juniper vMX Junos с помощью библиотеки ncclient

В предыдущих разделах приводились примеры использования библиотеки ncclient для взаимодействия с Cisco IOS-XE NETCONF API. В этом разделе рассматриваются примеры работы с Juniper Junos.

Для нашего примера в конфигурации Juniper vMX создано четыре статических маршрута.

Ниже приведены два результата, полученных непосредственно в интерфейсе командной строки с использованием команд `show` и `show | display xml` соответственно.

```
routing-options {
    static {
        route 0.0.0.0/0 next-hop 10.0.0.2;
        route 10.1.100.0/24 next-hop 10.254.1.1;
        route 10.2.200.0/24 next-hop 10.254.1.1;
        route 10.33.100.0/24 next-hop 192.168.1.1;
    }
}
```

Вывод в формате XML содержит родительский тег `<configuration>`, который здесь не показан в целях экономии места.

```
<routing-options>
    <static>
        <route>
            <name>0.0.0.0/0</name>
            <next-hop>10.0.0.2</next-hop>
        </route>
        <route>
            <name>10.1.100.0/24</name>
            <next-hop>10.254.1.1</next-hop>
        </route>
        <route>
            <name>10.2.200.0/24</name>
            <next-hop>10.254.1.1</next-hop>
        </route>
        <route>
            <name>10.33.100.0/24</name>
            <next-hop>192.168.1.1</next-hop>
```

Tlgm: @it_boooks


```
        </route>
    </static>
</routing-options>
```

Добавим новый статический маршрут, используя для этого библиотеку ncclient. На основе ранее полученных знаний можно разделить эту процедуру на несколько этапов, или шагов:

1. Создание нового экземпляра объекта устройства с помощью метода manager.connect().
2. Выбор требуемого типа операции: `merge`, `delete` или `replace`. Поскольку необходимо добавить новый статический маршрут, достаточно выполнить операцию по умолчанию `merge`.
3. Генерация требуемой строки в формате XML. Следует помнить о необходимости размещения строки внутри тега `<config>`, а поскольку этот пример демонстрирует работу с Junos, для генерации требуемой XML-строки (или объекта) весьма полезным будет использование конвейера `| xml` в интерфейсе командной строки.
4. Сохранение сформированной строки как новой переменной языка Python.
5. Выбор целевого хранилища данных конфигурации. Отметим, что Juniper поддерживает конфигурацию-кандидата, которой можно воспользоваться, чтобы определить, чем она отличается от изменяемых рабочих конфигураций.
6. Выполнение вызова API с использованием метода `edit_config()` созданного экземпляра объекта устройства.
7. Если конфигурация-кандидат использовалась как целевое хранилище данных, то передать ее как основную рабочую конфигурацию.

Новый добавляемый статический маршрут имеет префикс `172.16.20.0/24` и значение next-hop (следующий переход) `10.0.0.2`.

```
>>> vmx = manager.connect(host='junos-vmx', port=830, username='ntc',
...                       password='ntc123', hostkey_verify=False,
...                       device_params={}, allow_agent=False,
...                       look_for_keys=False)
...
>>>
>>> configuration = """
...     <config>
...       <configuration>
...         <routing-options>
...             <static>
...                 <route>
...                     <name>172.16.20.0/24</name>
...                     <next-hop>10.0.0.2</next-hop>
...                 </route>
...             </static>
...         </routing-options>
...       </configuration>
```

Tlgm: @it_boooks


```
...     </config>
... """
>>>
>>> response = vmx.edit_config(target='candidate', config=configuration)
>>>
>>> vmx.commit()
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1F4/junos"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:37c92cf6-f1d6-4af5-b813-1d3be0eccd99">
Setting up Virtual platform specific options
<ok/>
</rpc-reply>
>>>
```

Основное отличие этого примера состоит в том, что здесь особо выделено использование хранилища данных конфигурации-кандидата и последующее перемещение ее в основную рабочую конфигурацию.

Если необходимо удалить все неиспользуемые статические маршруты и оставить единственный статический маршрут, определенный по умолчанию, просто включите в XML-строку операцию `operation=replace` в правильной локации.

```
>>> configuration = """
...     <config>
...       <configuration>
...         <routing-options operation="replace">
...             <static>
...                 <route>
...                     <name>0.0.0.0/0</name>
...                     <next-hop>10.0.0.2</next-hop>
...                 </route>
...             </static>
...         </routing-options>
...       </configuration>
...     </config>
... """
>>>
>>> response = vmx.edit_config(target='candidate', config=configuration)
>>>
>>> vmx.commit()
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1F4/junos"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:dbc0c58b-4b3b-488f-89f2-67608b28cc77">
Setting up Virtual platform specific options
<ok/>
</rpc-reply>
>>>
```



В дополнение к тегу `<config>` Juniper также поддерживает XML-тег `<config-text>`, позволяющий передавать конфигурацию в виде обычного текста, размеченного фигурными скобками, непосредственно из командной строки.

Tlgm: @it_boooks

Внесение изменений в конфигурацию Cisco IOS-XR с помощью библиотеки ncclient

Когда мы начали подробно изучать работу с NETCONF, вероятно, вы обратили внимание на важность понимания того, как правильно формировать XML-объекты, чтобы успешно получить требуемые данные от устройства или правильно внести изменения в конфигурацию. Об этом было сказано неоднократно. Практически то же самое можно утверждать о необходимости знания отличий команд CLI в различных операционных системах и на устройствах различных производителей. Тем не менее существуют комитеты и рабочие группы, разрабатывающие модели данных, независимые от конкретных производителей оборудования. Одной из таких рабочих групп является OpenConfig Working Group (WG), занимающаяся разработкой независимой модели данных. На самом нижнем уровне это означает, что можно передавать один и тот же XML-объект на устройства от различных производителей, используя операцию `<edit-config>` для внесения изменений в конфигурацию.

В настоящее время некоторые производители оборудования добавляют поддержку независимых моделей данных от OpenConfig WG, но пока еще рано говорить о массовом внедрении таких моделей. Например, Juniper Junos и Cisco IOS-XR поддерживают модель данных OpenConfig BGP. Рассмотрим эту методику на примере Cisco IOS-XR.

Для маршрутизатора Cisco IOS-XR определена следующая базовая конфигурация BGP:

```
!
router bgp 65512
 bgp router-id 1.1.1.1
!
```

Используя NETCONF и модель OpenConfig BGP, представленную в формате XML, можно извлечь существующую конфигурацию BGP.

```
>>>
>>> get_filter = """
...     <bgp xmlns="http://openconfig.net/yang/bgp">
...       <global>
...         <config>
...         </config>
...       </global>
...     </bgp>
... """
>>>
>>> xr = manager.connect(host='ios-xrv', port=22, username='ntc',
...                      password='ntc123', hostkey_verify=False,
...                      device_params={}, allow_agent=False,
...                      look_for_keys=False)
>>>
>>> nc_get_reply = xr.get(('subtree', get_filter))
>>>
>>> print(etree.tostring(nc_get_reply.data, pretty_print=True))
```

Tlgm: @it_boooks


```
<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <bgp xmlns="http://openconfig.net/yang/bgp">
   <global>
    <config>
     <as>65512</as>
     <router-id>1.1.1.1</router-id>
    </config>
   </global>
  </bgp>
 </data>

>>>
```

Теперь, когда понятна структура возвращаемого объекта, воспользуемся ею как основой для создания объекта `<config>`, который изменяет идентификатор ID маршрутизатора.

```
>>> config = """
...     <config>
...      <bgp xmlns="http://openconfig.net/yang/bgp">
...       <global>
...        <config>
...         <router-id>10.10.10.10</router-id>
...        </config>
...       </global>
...      </bgp>
...     </config>
...  """
>>>
>>> response = xr.edit_config(target='candidate', config=config)
>>>
>>> xr.commit()
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:9e0f496b-0685-42ca-870b-6d0307b92641"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
```

По мере того как производители оборудования и операционные системы будут обеспечивать все более широкую поддержку независимых моделей данных, появится возможность выполнения точно такого же вызова API на разнообразных устройствах, то есть работа с различными типами устройств существенно упростится. Но пока такой возможности нет, вам необходимо хорошо понимать структуру XML-объектов, требуемых в каждой конкретной сетевой операционной системе.

Операции NETCONF, зависящие от конкретного производителя оборудования

В этом разделе основное внимание сосредоточено на двух наиболее часто используемых методах библиотеки ncclient: `<edit-config>` и `<get>`. Но, как вы

Tlgm: @it_boooks

уже могли заметить в предыдущих разделах текущей главы, при выполнении встроенной функции `dir()` по отношению к объекту устройства выводится длинный список других методов, о которых необходимо знать для успешного осуществления процесса автоматизации сети с помощью NETCONF и ncclient. Ниже приводятся краткие описания некоторых из этих методов:

- `close_session` – напомним, что NETCONF использует постоянное SSH-соединение. Этот метод позволяет разорвать соединение (завершить сеанс);
- `commit` – используется для перемещения конфигурации-кандидата в активную рабочую конфигурацию;
- `connected` – используется для проверки и установления наличия активного сеанса взаимодействия с устройством. Это атрибут, поэтому используется в форме `device.connected` и возвращает логическое значение (`True` или `False`);
- `copy_config` – создает или заменяет полностью все содержимое заданного хранилища данных конфигурации на содержимое другого хранилища данных конфигурации;
- `delete_config` – удаляет заданное хранилище данных конфигурации;
- `lock` и `unlock` – в реальной эксплуатационной среде может возникнуть необходимость блокировки хранилища данных конфигурации с целью его защиты от внесения каких-либо изменений посторонними лицами или системами во время вашего сеанса NETCONF. После завершения сеанса можно снять блокировку конфигурации;
- `server_capabilities` – объект, который можно просматривать в цикле, чтобы узнать обо всех функциональных возможностях, поддерживаемых сервером.

До сих пор при изучении Python-библиотеки ncclient главное внимание уделялось независимым от конкретного производителя способам выполнения операций, которые работают с любыми устройствами. При этом предполагалось, что пользователь хорошо знает, как правильно формировать XML-объекты. Но необходимо также понимать, что не все производители оборудования обеспечивают одинаковую реализацию NETCONF, несмотря на то что это протокол, определенный отраслевым стандартом. Например, некоторые производители оборудования создали собственные операции NETCONF для конкретных платформ или собственные специализированные методы для библиотеки ncclient, чтобы упростить выполнение наиболее востребованных операций. Подобные операции фактически конфликтуют со стандартными операциями NETCONF, такими как `<edit-config>`, `<get>`, `<lock>`, `<unlock>` и `<commit>`.

Этот факт можно подтвердить несколькими конкретными примерами:

- коммутаторы HPE Comware 7 поддерживают стандартные операции NETCONF, но предлагают также несколько специализированных операций для сохранения конфигурации, для отката (возврата) к другой конфигурации, при этом напрямую выполняются команды CLI-интерфейса. Специализированные операции соответствуют встроенным методам

Tlgm: @it_boooks

библиотеки ncclient. Например, для выполнения команд вывода необходимо пользоваться методом `cli_display()`, для сериализации данных конфигурации в сеансе NETCONF – методом `cli_config()`;

- для Juniper созданы специализированные методы в библиотеке ncclient, такие как `load_configuration()`, `get_configuration()`, `compare_configuration()` и `command()`, но этот список далеко не полон. Некоторые методы для Juniper представляют собой обертки для упрощения выполнения наиболее востребованных задач с применением стандартных операций NETCONF, другие используют специализированные для Juniper операции NETCONF RPC;
- устройство Cisco Nexus также поддерживает возможности передачи команд CLI-интерфейса, используя для этого метод `exec_command()`, который соответствует специализированной для Nexus операции NETCONF с именем `exec-command`.

Для работы с этими зависимыми от конкретного производителя оборудования операциями при использовании библиотеки ncclient необходимо явно определить правильную платформу в параметре `device_params` при создании экземпляра объекта устройства.

Во всех предыдущих примерах использовалось значение `device_params={}`, потому что выполнялись операции, определенные отраслевым стандартом, с помощью методов библиотеки ncclient. Если вы выбираете методы и операции, специализированные для конкретного производителя оборудования, то необходимо присвоить соответствующее значение параметру `device_params`.

Ниже приведено несколько примеров применения параметра `device_params` при выполнении специализированных для конкретного производителя оборудования операций и методов.

```
vmx = manager.connect(host='junos-vmx', port=830, username='ntc',
                      password='ntc123', hostkey_verify=False,
                      device_params={'name': 'junos'},
                      allow_agent=False, look_for_keys=False)

nxos = manager.connect(host='nxos1', port=22, username='ntc',
                      password='ntc123', hostkey_verify=False,
                      device_params='name': 'nexus'},
                      allow_agent=False, look_for_keys=False)

hpe = manager.connect(host='hpe5930', port=830, username='ntc',
                      password='ntc123', hostkey_verify=False,
                      device_params={'name': 'hpcomware'},
                      allow_agent=False, look_for_keys=False)
```

В этом разделе рассматривались основы практического использования Python-библиотек requests и ncclient, предназначенных для обмена данными с современными программными сетевыми API. В следующем разделе основное внимание будет сосредоточено на использовании протокола SSH в программах на языке Python, так как SSH все еще остается наиболее часто применяемым интерфейсом с сетевыми устройствами.

Tlgm: @it_boooks

Использование библиотеки netmiko

Команды CLI-интерфейса, передаваемые по SSH-соединению, фактически являются одним из основных способов управления сетевой инфраструктуры операторами и сетевыми инженерами. Команды передаются через постоянное SSH-соединение на сетевое устройство, затем устройство интерпретирует эти команды и возвращает ответ в виде текста, выводимого в окне терминала и удобного для чтения человеком. Сам по себе протокол SSH не поддерживает передачу структурированных данных, закодированных в форматах XML или JSON. Несмотря на то что SSH не является современным программируемым API, этот протокол очень важен для понимания того, каким образом можно использовать язык Python для автоматизации сетевых операций с помощью SSH, по трем причинам:

- не все устройства поддерживают программируемые API;
- возможность автоматизации внедрения какого-либо конкретного API;
- даже при автоматизации устройства с применением API:
  - необходимо иметь резервный план;
  - не все операции устройства могут поддерживаться выбранным API.

Это не оптимальное решение, поскольку оно демонстрирует несовершенство и недостаточную готовность API более низкого уровня.

В этом разделе рассматриваются основы практического использования широко распространенного SSH-клиента с открытым исходным кодом на языке Python – netmiko.

Главная цель netmiko – упрощение управления устройством через SSH-соединение, в наибольшей степени это относится к управлению сетевыми устройствами. В основу netmiko заложена другая библиотека с именем paramiko.

Здесь мы подробно рассматриваем именно netmiko, так как эта библиотека обеспечивает достаточно низкий входной уровень при ее освоении и предлагает готовые инструменты для обмена данными с многочисленными типами сетевых устройств. Netmiko поддерживает более двух десятков типов устройств, в том числе Arista, Brocade, Cisco, Dell, HPE, Juniper, Palo Alto Networks и многие другие. Еще одним замечательным свойством netmiko является полная идентичность ее использования на устройствах различных производителей. Это в определенной степени похоже на аналогичное свойство библиотеки ncclient. При работе с ncclient единственным различием между платформами был XML-объект, передаваемый на устройство. При работе с netmiko единственным различием между платформами являются команды, передаваемые на устройство.

Итак, начнем изучение netmiko.

Для установки библиотеки netmiko можно воспользоваться утилитой `pip`:

```
pip install netmiko
```

В первую очередь необходимо импортировать правильный объект устройства из библиотеки netmiko. Этот объект управляет установлением и настройкой SSH-соединения, завершением (разрывом) соединения и передачей

Tlgm: @it_boooks

команд на устройство. Аналогичный подход мы наблюдали при изучении библиотеки ncclient.

```
>>> from netmiko import ConnectHandler
>>>
```

Теперь можно установить SSH-соединение с сетевым устройством и создать netmiko-объект устройства. Объект `ConnectHandler` управляет SSH-соединением с заданным сетевым устройством.

```
>>> device = ConnectHandler(host='veos', username='ntc', password='ntc123',
                            device_type='arista_eos')
>>>
```

Здесь средствами языка Python с использованием библиотеки netmiko создано активное SSH-соединение с коммутатором Arista. Поскольку каждая платформа поддерживает различные команды и работает с протоколом SSH по-разному, обязательно требуется определение значения параметра `device_type` при создании экземпляра объекта `ConnectHandler`.

Теперь можно просмотреть список доступных методов для нового объекта устройства `device` с помощью встроенной функции `dir()`.

```
>>> dir(device)
[...некоторые методы удалены для краткости...,'check_config_mode', 'check_enable_mode',
'cleanup', 'clear_buffer', 'commit', 'config_mode', 'device_type',
'disable_paging', 'disconnect', 'enable', 'establish_connection',
'exit_config_mode', 'exit_enable_mode', 'find_prompt', 'global_delay_factor',
'host', 'key_file', 'key_policy', 'normalize_cmd', 'normalize_linefeeds',
'password', 'port', 'protocol', 'read_channel', 'read_until_pattern',
'read_until_prompt', 'read_until_prompt_or_pattern', 'remote_conn',
'remote_conn_pre', 'secret', 'select_delay_factor', 'send_command',
'send_command_expect', 'send_command_timing', 'send_config_from_file',
'send_config_set', 'session_preparation', 'set_base_prompt', 'set_terminal_width',
'special_login_handler', 'ssh_config_file', 'strip_ansi_escape_codes',
'strip_backspaces', 'strip_command', 'strip_prompt', 'system_host_keys',
'telnet_login', 'timeout', 'use_keys', 'username', 'verbose', 'write_channel']
>>>
```

Как сетевой инженер вы должны чувствовать себя достаточно уверенно, глядя на множество атрибутов, выведенных функцией `dir()`, поскольку почти все они имеют отношение к сетевой среде. Некоторые из этих атрибутов и методов мы рассмотрим более подробно в следующих подразделах.

Проверка промпта устройства

Для проверки строки промпта (приглашения к вводу команды) используется метод `find_prompt()`.

```
>>> device.find_prompt()
u'eos-spine1#'
>>>
>>> output = device.find_prompt()
```

Tlgm: @it_boooks


```
>>>
>>> print(output)
eos-spine1#
>>>
```

Переход в режим конфигурации

Поскольку netmiko обеспечивает работу с различными производителями оборудования и знает, что означает режим конфигурации, в библиотеке имеется метод для перехода в режим конфигурации (configuration mode), который работает с различными производителями оборудования. Разумеется, эта команда netmiko скрыто использует различные команды в зависимости от конкретной ОС.

```
>>> output1 = device.config_mode()
>>>
>>> output2 = device.find_prompt()
>>>
>>> print(output2)
eos-spine1(config)#
>>>
```

Передача команд на устройство

Наиболее частой операцией, выполняемой при работе с netmiko, является передача команд на устройство. Рассмотрим несколько методов, поддерживающих эту операцию.

Для простой передачи одной команды на устройство можно воспользоваться одним из следующих трех методов:

- `send_command_expect()` – метод используется для команд с длительным временем выполнения, то есть устройству потребуется много времени для обработки (например, при выполнении команд `show run` для крупных стоек с большим количеством устройств, `show tech` и т. п.). По умолчанию этот метод ожидает появления определенной строки промпта как признака завершения выполнения команды и только после этого возвращает результат. Дополнительно можно передать новую строку промпта, которую должен ожидать метод, в зависимости от передаваемой команды;
- `send_command_timing()` – метод используется для команд с небольшим временем выполнения. Основан на оценке временного интервала и не проверяет наличие строки промпта;
- `send_command()` – это более старый метод в библиотеке netmiko, поэтому в настоящее время он работает как обертка для упрощенного вызова метода `send_command_expect()`. Таким образом, `send_command()` и `send_command_expect()` выполняют одну и ту же операцию.

Рассмотрим несколько примеров.

Пример сбора информации с помощью команды `show run` и последующего вывода первых 165 символов данных для проверки:

Tlgm: @it_boooks


```
>>> output = device.send_command('show run')
>>>
>>> print(output[:165])
! Command: show running-config
! device: eos-spine1 (vEOS, EOS-4.15.2F)
!
! boot system flash:vEOS-lab.swi
!
transceiver qsfp default-mode 4x10G
!
hostname eos-spine
>>>
```

Передача команды, изменяющей строку промпта (напомним, что мы остаемся в режиме конфигурации после входа в него с помощью команды `output1 = device.config_mode()`), выполняется следующим образом:

```
>>> output = device.send_command_expect('end')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/dist-packages/netmiko/base_connection.py",
line 681, in send_command_expect
    return self.send_command(*args, **kwargs)
  File "/usr/local/lib/python2.7/dist-packages/netmiko/base_connection.py",
line 673, in send_command
    search_pattern))
IOError: Search pattern never detected in
send_command_expect: eos\-spine1\(config\)\#
>>>
```

Вывод трассировки стека вполне предсказуем, так как метод `send_command_expect()` ожидает появления строки промпта, заданной по умолчанию. Поскольку мы находимся в режиме конфигурации с текущей строкой промпта `eos-spine1(config)#` при вводе команды `end`, новой строкой промпта должна быть строка `eos-spine1#`.

Для правильного выполнения команды, изменяющей строку промпта, существуют два варианта. Во-первых, можно воспользоваться параметром `expect_string`, который определяет новую, правильно интерпретируемую строку промпта.

```
>>> output = device.send_command_expect('end', expect_string='eos-spine1#')
>>>
```

Во-вторых, можно воспользоваться методом `send_command_timing()`, который основан на оценке временного интервала и не ожидает появления конкретной строки промпта как признака завершения выполнения.

```
>>> output = device.send_command_timing('end')
>>>
```

Мы рассмотрели три метода передачи одиночных простых команд из библиотеки netmiko. Теперь рассмотрим еще два полезных метода, позволяющих одновременно передавать сразу несколько команд.

Tlgm: @it_boooks

Библиотека netmiko предлагает метод `send_config_set()`, принимающий параметр, который необходимо обрабатывать итеративно (то есть в цикле). В приведенном ниже примере используется список языка Python, но вы также можете воспользоваться множеством (set) языка Python.

```
>>> commands = ['interface Ethernet5', 'description configured by netmiko', 'shutdown']
>>>
>>> output = device.send_config_set(config_commands=commands)
>>>
>>> print(output)
config term
eos-spine1(config)#interface Ethernet5
description configured by netmiko
eos-spine1(config-if-Et5)#description configured by netmiko
eos-spine1(config-if-Et5)#shutdown
eos-spine1(config-if-Et5)#end
eos-spine1#
>>>
```

Метод проверяет, был ли уже выполнен переход в режим конфигурации. Если ответ отрицательный, метод сам осуществляет вход в режим конфигурации, выполняет указанные команды и по умолчанию выходит из режима конфигурации. Это можно проверить, внимательно изучив вывод результата выполнения, показанный в предыдущем примере.

Еще один метод обеспечивает выполнение команд из файла. Это позволяет выполнять такие операции, как создание шаблона Jinja, обработка с его помощью различных наборов данных, запись данных в файл. Весь этот набор команд можно записать в файл, а затем выполнить с помощью метода `send_config_from_file()` из библиотеки netmiko. Вспомнив все, что мы изучали в главах 4 и 6, рассмотрим, как можно выполнить такую рабочую последовательность команд.

```
>>> from netmiko import ConnectHandler
>>> from jinja2 import Environment, FileSystemLoader
>>>
>>> device = ConnectHandler(host='veos', username='ntc', password='ntc123',
                            device_type='arista_eos')
>>>
>>> interface_dict = {
...     "name": "Ethernet6",
...     "description": "Server Port",
...     "vlan": 10,
...     "uplink": False
... }
>>>
>>> ENV = Environment(loader=FileSystemLoader('.'))
>>> template = ENV.get_template("config.j2")
>>> commands = template.render(interface=interface_dict)
>>>
>>> with open('veos.conf', 'w') as config_file:
...     config_file.writelines(commands)
...
```

Tlgm: @it_boooks


```
>>> output = device.send_config_from_file('veos.conf')
>>>
>>> verification = device.send_command('show run interface Eth6')
>>>
>>> print(verification)
interface Ethernet6
   description Server Port
   switchport access vlan 10
>>>
```



Все показанное в предыдущем примере уже подробно рассматривалось в предыдущих главах. Отметим, что для выполнения этого примера необходимо создать файл config.j2, содержащий шаблон Jinja, и разместить его в том каталоге, из которого производится вход в интерпретатор Python. Содержимое шаблона взято из аналогичного примера, рассматриваемого в главе 6, и выглядит следующим образом:

```
interface {{ interface.name }}
 description {{ interface.description }}
 switchport access vlan {{ interface.vlan }}
 switchport mode access
```

После выполнения всех необходимых операций с помощью библиотеки netmiko можно корректно отключиться от устройства (завершить соединение) с помощью метода `disconnect()`.

```
>>> device.disconnect()
>>>
```



Библиотека netmiko также используется в качестве основного SSH-драйвера для различных устройств в NAPALM, надежной в эксплуатации и поддерживающей многочисленных производителей сетевого оборудования библиотеке на языке Python, предназначенной для конфигурирования устройств и извлечения данных. Библиотека NAPALM рассматривается в приложении Б.

На этом изучение процесса автоматизации сетевых устройств на основе SSH-соединения с использованием библиотеки netmiko завершено. Мы подробно рассмотрели процедуры автоматизации разнообразных типов сетевых устройств с применением различных типов API. Это поможет вам реализовать процесс автоматизации вне зависимости от того, с какими устройствами и API вам придется работать.

Резюме

В этой главе все внимание было сосредоточено на двух основных типах API, используемых в сетевой среде: API на основе протокола HTTP и NETCONF. Были кратко описаны основы организации и функционирования API, рассматривалось практическое применение API в интерфейсе командной строки и с помощью инструментальных средств с графическим пользовательским интерфейсом. Демонстрировались процедуры автоматизации сетевых устройств с использованием тех же API в программах на языке Python. Мы подробно

Tlgm: @it_boooks

изучили RESTful API на основе HTTP и API, не поддерживающие концепции REST. При практическом применении разнообразных типов API на основе протокола HTTP необходимо всегда помнить о следующих фактах:

- не все API на основе HTTP поддерживают принципы REST, но используемые для обоих типов API инструментальные средства одинаковы;
- все рассмотренные в этой главе RESTful API используют HTTP в качестве транспортного протокола;
- API на основе HTTP могут использовать форматы XML или JSON для кодирования данных, но устройство может реализовать только один из вариантов. Автор API определяет, какой из форматов кодировки поддерживается;
- RESTCONF – это реализация RESTful API, в которой используется протокол HTTP для доступа и обработки данных, определяемых моделями YANG;
- инструментальные средства, подобные cURL и Postman, удобны и полезны, когда вы только начинаете освоение API, но для написания кода, взаимодействующего с API на основе HTTP, потребуется специализированная библиотека с поддержкой протокола HTTP, например библиотека requests, написанная на языке Python;
- необходимо уделить особое внимание правильному использованию ключевых слов HTTP, применяемых при внесении изменений в конфигурацию, – использование неверного ключевого слова может привести к весьма нежелательным последствиям;
- необходимо пользоваться документацией на API или инструментальными средствами для устройств (Arista Command Explorer, NX-API Developer Sandbox, ASA API Documentation & Console), чтобы глубоко понимать, как формировать правильный API-запрос: URL, заголовки, HTTP-метод и тело.



В этой главе мы не рассматривали специализированную реализацию API для SDN Controller, поскольку процесс изучения нового API практически одинаков. Настоятельно рекомендуется изучать документацию на API, чтобы понять, каким образом сформирован и реализован программный интерфейс для этой платформы. Затем необходимо начать практическое освоение API, используя для этого Postman или Python. В этой главе нашей целью было подробное рассмотрение примеров обмена данными с различными платформами, чтобы изучить наиболее общие принципы использования API. Но в дальнейшем вы должны продолжить изучение практического применения API для других сетевых устройств (физических или виртуальных), контроллеров или даже облачных платформ, которые также используют API на основе протокола HTTP.

В этой главе также рассматривалось использование NETCONF и совместная работа NETCONF с Python-библиотекой ncclient. Применяя NETCONF на практике, необходимо всегда помнить о следующих фактах:

- NETCONF – это протокол, основанный на установлении постоянного соединения и создании сеанса обмена данными, который в основном использует SSH как транспортный протокол;

Tlgm: @it_boooks

- для NETCONF необходимо обеспечить обмен данными между клиентом и сервером до того, как клиент получит возможность выполнить требуемый запрос;
- NETCONF поддерживает только кодировку XML;
- NETCONF не зависит от того, каким образом смоделированы передаваемые данные;
- документы XML и объекты, представляющие результаты, являются XML-представлениями данных, которые соответствуют схеме или модели данных для целевого устройства. NETCONF может использовать данные в кодировке XML, соответствующие данным, смоделированным с помощью XML Schema Definitions (XSDs), YANG или любого другого языка либо схемы.

В конце главы подробно рассматривалась автоматизация сетевых устройств с использованием протокола SSH и библиотеки netmiko.

Когда вы будете самостоятельно продолжать процесс автоматизации сетевых устройств с помощью различных типов API, помните, что в этом нет никакого «волшебства» – необходимо работать профессионально и хорошо понимать, как следует использовать тот или иной API.

Tlgm: @it_boooks

Глава 8

Управление исходным кодом с помощью Git

До настоящего момента в этой книге было продемонстрировано множество способов включения процедур автоматизации в вашу учебную сетевую среду («песочницу») – с использованием скриптовых языков, подобных Python (глава 4), или языков моделирования, таких как Jinja (глава 6). Постоянное расширение применения скриптов на языке Python или шаблонов на языке Jinja означает, что управление этими артефактами (в текущем контексте под артефактами подразумеваются файлы, содержащие созданные скрипты, шаблоны и прочие средства автоматизации, которые вы используете на практике) становится весьма важной задачей. Особенно важным становится управление изменениями, вносимыми в содержимое всех этих артефактов (немного позже эта важность будет обоснована).

В этой главе подробно рассматриваются методики практического применения инструментального средства управления исходным программным кодом (source control), то есть инструмента, специально предназначенного для управления артефактами, которые вы создаете и используете в процессах автоматизации сети. Инструментальное средство управления исходным кодом позволяет избежать беспорядка и потенциальных ошибок, связанных с более примитивными подходами, такими как добавление меток даты и времени в имена файлов, а также обеспечит защиту от случайного удаления или перезаписи файлов.

В начале главы объясняется сама идея управления исходным кодом в обобщенном виде. Далее во всех подробностях рассматривается конкретное инструментальное средство управления исходным кодом – Git. Тем не менее общие свойства и характеристики, обсуждаемые в следующем разделе, не являются присущими какому-либо конкретному инструменту управления исходным кодом.

Tlgm: @it_boooks

Варианты использования средств управления исходным кодом

Проще говоря, управление исходным кодом (source control) – это способ (методика) отслеживания состояния файлов и вносимых в них изменений во времени. Такую методику также называют управлением версиями (version control), или управлением изменениями (revision control). Здесь приведено самое обобщенное определение, поэтому рассмотрим некоторые конкретные варианты использования:

- разработчик, пишущий код как часть крупного проекта создания ПО, должен отслеживать общее состояние исходного кода с помощью соответствующих инструментов. Это наиболее часто встречающийся и широко известный вариант использования, поэтому большинство людей при упоминании концепции управления исходным кодом в первую очередь вспоминает именно этот вариант;
- член команды администраторов, управляющих сетевыми устройствами. Предоставляется возможность хранить файлы конфигурации устройств и отслеживать изменения в них с помощью инструментальных средств управления исходным кодом;
- предположим, что вы отвечаете за сопровождение документации по некоторым частям IT-инфраструктуры в своей организации. Эту задачу также можно решить с помощью инструментальных средств управления исходным кодом.

В каждом из описанных выше вариантов управление исходным кодом – это отслеживание состояния файлов (сетевых конфигураций, документации, программного кода). Под отслеживанием состояния (tracking) подразумевается, что инструментальное средство управления исходным кодом сохраняет первоначальное содержимое файлов, все изменения, которые со временем вносятся в эти файлы, а также авторов всех внесенных изменений. Если внесение какого-либо изменения в один из отслеживаемых файлов приводит к нарушению общей согласованности и логической связности всей системы, то можно вернуться (откатиться – roll back) к предыдущей версии файла, отменив внесенные изменения и восстановив его гарантированное корректное состояние. В некоторых случаях (в зависимости от конкретного используемого инструментального средства) система управления исходным кодом может поддерживать совместную работу нескольких пользователей в распределенной среде.

Преимущества системы управления исходным кодом

В предыдущем разделе были отмечены некоторые из преимуществ использования инструмента управления исходным кодом. Здесь преимущества системы управления исходным кодом описываются более подробно.

Tlgm: @it_boooks

Отслеживание изменений

Самое главное преимущество состоит в том, что пользователю предоставляется возможность отслеживания во времени всех изменений, которые вносятся в файлы, хранящиеся в системе управления исходным кодом. Можно посмотреть состояние всех файлов в любой заданный момент времени, таким образом, относительно просто можно определить, что именно было изменено. Это преимущество часто недооценивают. При работе с большими файлами сетевой конфигурации весьма полезно иметь возможность точно определять, что изменилось в новой версии, по сравнению с предыдущей. Кроме того, большинство инструментальных средств управления исходным кодом предоставляет еще и возможность добавления метаданных о каждом вносимом изменении, как, например, причину внесения изменения, ссылку (указание) на источник изменения или талон неисправности (trouble ticket). Такие дополнительные метаданные могут оказаться чрезвычайно полезными при поиске и устранении возникших проблем.

Учетные записи

Инструменты управления исходным кодом отслеживают не только внесение изменений, но также следят за тем, кто вносит эти изменения. Каждое изменение фиксируется с указанием учетной записи, от имени которой было внесено изменение. Это особенно удобно при работе группы в распределенной среде, когда несколько членов группы могут одновременно управлять сетевыми конфигурациями или файлами конфигурации сервера. При таком подходе никогда не возникает вопрос: «Кто внес это изменение?» Сама система управления исходным кодом даст исчерпывающий ответ.

Процесс и рабочий поток

Использование инструментальных средств управления исходным кодом поможет вам и вашей организации создать правильный процесс и рабочий поток (workflow). Более подробно это будет рассматриваться в главе 10, а сейчас мы просто отметим необходимость строгой фиксации (записи в журнал) всех изменений в системе управления исходным кодом перед внедрением их в реальную эксплуатацию. При этом создается линейная хронология изменений в журнале (log) с указанием конкретного лица (учетной записи), ответственного за внесение каждого набора изменений. Подобный подход позволяет выполнять такие операции, как обзор или инспекция (другой сотрудник может проверить вносимые вами изменения перед внедрением их в реальную эксплуатацию) либо тестирование (создание автоматизированных тестов, проверяющих файлы в системе управления исходным кодом).

Tlgm: @it_boooks

Преимущества системы управления исходным кодом в сетевой среде

Несмотря на то что концепцию управления исходным кодом обычно связывают с разработкой ПО, система управления исходным кодом предлагает немалые преимущества и для специалистов по сетям. Вот несколько примеров:

- скрипты на языке Python (которые могут писать читатели этой книги), взаимодействующие с сетевыми устройствами, могут быть размещены в системе управления исходным кодом, чтобы упростить управление версиями каждого скрипта;
- файлы конфигурации сетевых устройств можно разместить в системе управления исходным кодом. Это позволит отслеживать состояние конфигурации каждого сетевого устройства в любой момент времени. Широко известное инструментальное средство RANCID применяет такой подход для хранения резервных копий файлов конфигурации сетевых устройств;
- существенно упрощается выявление изменений и различий между версиями конфигураций сетевых устройств. Это позволяет без затруднений проверить тот факт, что внесены только требуемые изменения (например, что по неосторожности не удалена виртуальная сеть VLAN из некорректно сформированного основного канала 802.1Q);
- шаблоны сетевой конфигурации можно разместить в системе управления исходным кодом. При этом предоставляется возможность отслеживания изменений в шаблонах перед их использованием для генерации файлов конфигурации сетевых устройств или отчетов;
- систему управления исходным кодом можно использовать для ведения сетевой документации;
- любые изменения во всех перечисленных выше типах файлов строго фиксируются с указанием лица (учетной записи), ответственного за внесение этих изменений, таким образом, исключаются случаи «коллективной ответственности» и взаимные обвинения.

После краткого описания премуществ системы управления исходным кодом и пользы, которую она может принести любой организации и рабочему процессу с точки зрения автоматизации сети, мы переходим к подробному рассмотрению конкретного инструментального средства управления исходным кодом, которое весьма широко применяется на практике: Git (https://git-scm.com/).

Знакомство с Git

Git – это самый новый программный продукт в длинном списке инструментальных средств управления исходным кодом, который появился как эффективный инструмент управления версиями для большинства проектов с открытым

Tlgm: @it_boooks

исходным кодом. (Следует особо отметить, что с помощью Git осуществляется управление исходным кодом ядра Linux.) Именно поэтому из обширного набора средств управления исходным кодом мы выбрали Git, тем не менее следует помнить о том, что существуют и другие инструменты управления исходным кодом. К сожалению, мы не можем уделить им внимание в данной книге.

Для начала попробуем понять, почему появился инструмент под названием Git и как он развивался.

Краткая история создания и развития Git

Как уже было сказано выше, Git – это инструментальное средство управления исходным кодом, которое, в частности, применяется для управления исходным кодом ядра Linux. Проект был создан Линусом Торвальдсом (Linus Torvalds), автором ядра Linux, в начале апреля 2005 года как реакция на возникший конфликт между сообществом разработчиков ядра Linux и владельцами проприетарной системы управления версиями, которая в то время использовалась сообществом (это была система BitKeeper).

При проектировании Торвальдс определил главные цели создаваемой системы Git:

- скорость – система должна обеспечивать быстрое внесение изменений («заплат» – patches) в исходный код ядра Linux;
- простота – проектное решение системы Git должно быть настолько простым, насколько это возможно;
- полная поддержка нелинейного процесса разработки – разработчикам ядра Linux требовалась система, которая могла бы обрабатывать множество параллельных ветвей (версий). Таким образом, новая система Git должна была поддерживать быстрое разветвление (branching) и объединение (merging), а сами ветви (branches) должны быть максимально простыми и удобными в управлении;
- поддержка работы в полностью распределенной среде – каждому разработчику требовалась полная копия всего исходного кода и хронология его изменений;
- масштабируемость – Git должен обладать способностью гибкого масштабирования, достаточного для работы с крупными проектами, такими как ядро Linux.

Разработка Git была выполнена в очень быстром темпе. В течение нескольких дней Git получил возможность «самообслуживания» (то есть исходный код программы Git уже управлялся самой системой Git). Через пару недель было выполнено первое объединение (merge) нескольких ветвей разработки. В конце апреля, то есть через несколько недель после начала работы над проектом, был выполнен эталонный тест Git в процессе применения патчей (patches) в дереве исходного кода ядра Linux с результатом 6.7 патча в секунду. В июне 2005 года система Git полностью управляла выпуском версии 2.6.12 ядра Linux, а выпуск версии 1.0 самой системы Git состоялся в конце декабря 2005 года.

Tlgm: @it_boooks

Во время написания этой книги самой свежей была версия Git 2.13.1. Версии Git доступны для всех основных десктоповых операционных систем (Linux, Windows, macOS). Самыми крупными и широко известными проектами с открытым исходным кодом, которые используют Git, являются ядро Linux (это было отмечено выше), Perl, настольная пользовательская графическая среда Gnome, Android, KDE Project, а также реализация X.Org графической системы X Window System. Кроме того, некоторые весьма широко используемые онлайновые сервисы, осуществляющие управление исходным кодом, основаны на Git, в том числе GitHub (https://github.com), BitBucket (https://bitbucket.org) и GitLab (http://about.gitlab.com). Некоторые из этих сервисов предлагают также локальные реализации, которые могут автономно работать в системе пользователя. Более подробно о системе GitLab можно будет узнать в главе 10, когда мы будем рассматривать концепцию непрерывной интеграции (continuous integration).

Терминология Git

Прежде чем приступить к подробному изучению Git, необходимо убедиться в том, что мы правильно понимаем соответствующие термины. Некоторые из терминов уже использовались ранее, но для полноты они также включены в следующий список:

- *репозиторий (repository)* – в системе Git это имя, которое присваивается базе данных, содержащей всю информацию о проекте (файлы и метаданные), а также хронологию изменений. (Здесь термин проект (project) используется для обозначения произвольной группы файлов, объединенных для решения конкретной задачи или выполнения конкретной работы.) Репозиторий представляет собой полную копию всех файлов и всей информации, связанной с проектом на протяжении его жизненного цикла. Важно отметить, что после добавления в репозиторий данные становятся неизменяемыми, то есть их нельзя редактировать после добавления. Это не означает, что в файлы, хранящиеся в репозитории, нельзя вносить изменения, это свидетельствует о том, что репозиторий хранит и отслеживает все файлы таким способом, что при внесении изменений в какой-либо файл создается новая запись в репозитории (в частности, Git использует SHA-хеширование для создания адресуемых по содержимому объектов в репозитории);
- *рабочий каталог (working directory)* – это каталог, в котором находится пользователь системы Git с целью редактирования файлов, содержащихся в репозитории. Отметим, что термин рабочий каталог также используется для других целей в операционных системах Linux/Unix/macOS (для обозначения текущего каталога, как выходные данные команды `pwd`). В системе Git рабочий каталог – это не то же самое, что текущий каталог, это сугубо специфическое для Git обозначение каталога, в котором хранится репозиторий *.git*;

Tlgm: @it_boooks

- *индекс (index)* – описание структуры каталогов репозитория и его содержимого в некоторый момент времени. Индекс представляет собой динамический бинарный файл, сопровождаемый системой Git и обновляемый при каждом внесении изменений пользователем и передаче (коммите) их в репозиторий;
- *коммит (commit)* – запись в репозитории Git, хранящая метаданные для каждого изменения, внесенного в репозиторий. Метаданные включают автора, дату коммита и сообщение (описание изменения, внесенного в репозиторий). Кроме того, коммит фиксирует состояние всего репозитория на момент выполнения самого коммита (то есть на момент внесения и подтверждения соответствующего изменения). Необходимо помнить, что когда говорят о «внесении изменения в репозиторий», это может означать внесение нескольких изменений в несколько файлов. Git позволяет объединить изменения в нескольких файлах в единый коммит. (Более подробно такая процедура будет рассматриваться в этой главе.)

Обзор архитектуры Git

Изучив смысл терминов, описанных в предыдущем разделе, можно перейти к обзору архитектуры Git. Здесь при рассмотрении архитектуры мы ограничимся только относительно высоким уровнем, но все же достаточно подробным для того, чтобы понять, как работает система Git.



Для более глубокого изучения архитектуры Git рекомендуется книга издательства O'Reilly «Version Control with Git, Second Edition» (http://shop.oreilly.com/product/0636920022862.do).

Как уже было отмечено выше, репозиторий (repository) Git – это база данных, содержащая всю информацию о проекте: файлы, входящие в проект, изменения, вносимые в проект со временем, метаданные, соответствующие этим изменениям (кто внес изменения, когда и т. д.). По умолчанию эта информация хранится в каталоге *.git* в корне вашего рабочего каталога (это можно изменить). Ниже приведен пример списка файлов только что созданного и инициализированного рабочего каталога репозитория Git, в котором показан подкаталог *.git*, где хранятся реальные данные:

```
relentless:npab-examples slowe (master)$ ls -la
total 0
drwxr-xr-x   3 slowe  staff  102 May 11 15:37 .
drwxr-xr-x  16 slowe  staff  544 May 11 15:37 ..
drwxr-xr-x  10 slowe  staff  340 May 11 15:37 .git
relentless:npab-examples slowe (master)$
```

Из приведенного примера видно, что команда вывода списка файлов выполнена в текущем каталоге *npab-examples*. В данном случае рабочим каталогом (working directory) является каталог *npab-examples*, а репозиторий Git обозначается как *npab-examples/.git*. Именно поэтому ранее было отмечено, что

Tlgm: @it_boooks

рабочий каталог и репозиторий – это не одно и то же. Новички часто путают рабочий каталог с репозиторием, но всегда необходимо помнить, что настоящий репозиторий располагается в подкаталоге *.git*.

В каталоге *.git* размещены разнообразные компоненты, из которых, собственно, и состоит репозиторий Git:

- индекс – выше он был определен как представление структуры и содержимого каталога репозитория в конкретный момент времени. Индекс расположен в файле *.git/index*;
- файлы, размещенные в репозитории Git, интерпретируются как адресуемые по содержимому объекты и хранятся в подкаталогах в *.git/objects*;
- все подробности, относящиеся к конфигурации репозитория, размещены в *.git/config*;
- метаданные о репозитории, изменения, хранящиеся в репозитории, и объекты репозитория можно найти в *.git/logs*.

Вся информация, хранящаяся в каталоге *.git*, сопровождается и обрабатывается самой системой Git – пользователю никогда не придется напрямую взаимодействовать с содержимым этого каталога. На протяжении всей текущей главы будут рассматриваться разнообразные команды, необходимые для работы с репозиторием: добавление файлов, коммит (внесение и подтверждение) изменений, отмена изменений и т. д. Поэтому сейчас мы переходим в следующий раздел, в котором подробно рассматривается практическая работа с системой Git.

Работа с системой Git

После первоначального ознакомления с архитектурой Git можно перейти к более практической задаче: реальной работе с системой Git.

В процессе освоения Git будет рассматриваться исключительно практический пример (во всяком случае, мы надеемся, что он окажется полезным с практической точки зрения). Предположим, что вы сетевой инженер, ответственный за развертывание и ввод в эксплуатацию некоторых инструментальных средств автоматизации сети в вашей сетевой среде. В процессе работы вам придется создавать скрипты на языке Python, шаблоны на языке Jinja и прочие файлы. Для управления этими файлами выбрано инструментальное средство Git, поэтому вы можете воспользоваться всеми многочисленными преимуществами системы управления исходным кодом.

В следующих разделах будут подробно рассматриваться основные этапы практического использования системы Git для управления файлами, созданными как часть рабочего процесса автоматизации сети.

Установка системы Git

Процедура установки системы Git чрезвычайно подробно и хорошо документирована, поэтому здесь она не описывается. Чаще всего система Git устанав-

Tlgm: @it_boooks

ливается в процессе установки разнообразных дистрибутивов Linux, но даже если это не так, то дистрибутивный пакет Git почти всегда доступен для установки с помощью менеджера пакетов многих дистрибутивов Linux (например, dnf для RHEL/CentOS/Fedora или apt для Debian/Ubuntu). Также доступны программы установки Git для ОС Windows и macOS, с помощью которых можно без затруднений установить Git в своей системе. Подробнейшие инструкции и все детали и варианты установки системы Git также доступны на веб-сайте Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

Создание репозитория

После установки системы Git в первую очередь необходимо создать репозиторий. Сначала создается каталог, в котором будет размещен репозиторий. Предположим, что вы используете ОС Ubuntu Linux, тогда эта процедура может выглядеть следующим образом (в других дистрибутивах Linux и в macOS команды должны быть почти такими же или даже точно такими же):

```
vagrant@trusty:~$ mkdir ~/net-auto
vagrant@trusty:~$
```

Затем нужно перейти во вновь созданный каталог и создать новый пустой репозиторий с помощью команды `git init`:

```
vagrant@trusty:~$ cd net-auto
vagrant@trusty:~/net-auto$ git init
Initialized empty Git repository in /home/vagrant/net-auto/.git
vagrant@trusty:~/net-auto$
```

Команда `git init` отвечает за инициализацию или создание нового репозитория Git. В результате ее выполнения создается каталог *.git*, а также все подкаталоги и прочее содержимое, необходимое для работы.



Различные команды `git`, рассматриваемые в этой главе, почти не отличаются друг от друга во всех системах, в которых может работать система Git. В следующих примерах используются разные дистрибутивы Linux (распознаваемые по отличающимся промптам командной оболочки), но применение Git в macOS практически ничем не отличается от применения Git в Linux. Работа с Git в Windows выполняется схожим образом, тем не менее возможны некоторые синтаксические особенности из-за различий в самих операционных системах.

Если сейчас выполнить команду `ls -la` в каталоге *net-auto*, то можно увидеть подкаталог *.git*, в котором хранится пустой репозиторий Git, созданный командой `git init`. Этот репозиторий готов к приему содержимого. Содержимым будут добавляемые файлы.

Добавление файлов в репозиторий

Добавление файлов в репозиторий представляет собой многоэтапный процесс:

1. Добавление файлов в рабочий каталог репозитория.
2. Запись файлов в индекс репозитория.

Tlgm: @it_boooks

3. Коммит (передача с подтверждением) занесенных в индекс файлов в репозиторий.

Вернемся к нашему примеру. Ранее был создан новый репозиторий Git для хранения файлов, являющихся частью проекта автоматизации сети. В первую очередь необходимо добавить в репозиторий файлы текущей конфигурации сетевых устройств. Допустим, что имеются три файла *sw1.txt*, *sw2.txt* и *sw3.txt*, которые содержат текущие конфигурации для трех коммутаторов.

Сначала эти файлы копируются в рабочий каталог (в нашем примере это каталог */home/vagrant/net-auto*). И снова напомним, что в общем случае рабочий каталог – это родительский каталог для подкаталога *.git* (который содержит действительный репозиторий Git). В системах Linux или macOS копирование файлов можно выполнить с помощью команды `cp`, в системе Windows для этого используется команда `copy`.

Теперь файлы находятся в рабочем каталоге, но пока еще не добавлены в сам репозиторий. Это означает, что система Git не отслеживает состояние таких файлов и их содержимого, следовательно, пока еще невозможно контролировать изменения, узнать, кто их автор, или откатиться к предыдущей версии.

Это состояние можно проверить, выполнив команду `git status`, которая для рассматриваемого примера выводит следующую информацию:

```
vagrant@trusty:~/net-auto$ git status
On branch master
(В главной ветви)

Initial commit
(Начальный коммит)

Untracked files:
(Неотслеживаемые файлы:)
  (use "git add <file>..." to include in what will be committed)
  (используйте команду "git add <file>...", чтобы внести в репозиторий файлы с помощью
коммита)

    sw1.txt
    sw2.txt
    sw3.txt

nothing added to commit but untracked files present (use "git add" to track)
(ничего не добавлено с помощью коммита, но неотслеживаемые файлы существуют (используйте
команду "git add", чтобы сделать их отслеживаемыми))
```

Вывод команды `git status` сообщает, что в рабочем каталоге находятся неотслеживаемые файлы, но в репозиторий ничего не добавлено. В выведенной информации содержится рекомендация: использовать команду `git add` для добавления перечисленных неотслеживаемых файлов в репозиторий, как показано ниже:

```
vagrant@trusty:~/net-auto$ git add sw1.txt
vagrant@trusty:~/net-auto$ git add sw2.txt
vagrant@trusty:~/net-auto$ git add sw3.txt
vagrant@trusty:~/net-auto$
```

Tlgm: @it_boooks

Можно воспользоваться некоторыми функциональными свойствами командной оболочки, чтобы добавить несколько файлов одной командой. Например, команда `git add sw*.txt` позволяет добавить сразу все три файла конфигурации коммутаторов в репозиторий.

После добавления файлов в область подтверждения (staging area) можно снова воспользоваться командой `git status`, чтобы узнать текущее состояние репозитория:

```
vagrant@trusty:~/net-auto$ git status
On branch master

Initial commit

Changes to be committed:
(Необходимо выполнить коммит изменений:)
  (use "git rm --cached <file>..." to unstage)
  (используйте команду "git rm --cached <file>..." для отмены подтверждения)

    new file:   sw1.txt
    new file:   sw2.txt
    new file:   sw3.txt

vagrant@trusty:~/net-auto$
```

На этот момент указанные файлы внесены в индекс Git. Это означает, что индекс Git синхронизирован с рабочим каталогом. С технической точки зрения эти файлы были добавлены как объекты, содержащиеся в объекте хранилища Git, но в текущем интервале времени пока еще не существует ссылок на эти объекты. Чтобы создать текущую корректную ссылку на объект, необходимо выполнить коммит (commit) (то есть передачу с подтверждением) изменений, внесенных в индекс.

Выполнение коммита изменений в репозиторий

Перед коммитом изменений в репозиторий необходимо выполнить несколько предварительных операций. Напомним, что одним из преимуществ использования Git в качестве инструментального средства управления исходным кодом является не только возможность отслеживания изменений в файлах, хранящихся в репозитории, но и возможность точно знать, кто внес каждый набор изменений. Чтобы эта информация всегда была доступной, сначала необходимо предоставить ее системе Git. (Это можно сделать сразу после установки системы Git, создавать новый репозиторий не обязательно.)

Внесение информации о пользователе в систему Git

Для системы Git существует набор параметров конфигурации, некоторые из которых предназначены для репозитория, другие – для пользователей, отдельная группа параметров – для всей системы. Выше уже отмечалось, что предназначенные для репозитория параметры конфигурации хранятся в файле *.git/config*. В нашем примере необходимо определить имя пользователя и адрес

Tlgm: @it_boooks

электронной почты (email), чтобы система Git могла отслеживать авторов каждого набора изменений, поэтому требуется редактирование не параметров конфигурации репозитория, а параметров конфигурации пользователей.

Но где хранятся значения этих параметров? Их можно найти в файле *.gitconfig*, размещенном в вашем домашнем каталоге. Это файл INI-типа, и с ним можно работать в любом текстовом редакторе по вашему выбору или с помощью команды `git config`. Для наглядности здесь демонстрируется использование команды `git config` с целью определения параметров пользователя.

Для определения имени и адреса электронной почты пользователя выполняются следующие команды:

```
vagrant@jessie:~/net-auto$ git config --global user.name "John Smith"
vagrant@jessie:~/net-auto$ git config --global user.email "john.smith@networktocode.com"
vagrant@jessie:~/net-auto$
```

Ключ `--global` здесь используется для определения значения, относящегося к общей конфигурации пользователей. Если необходимо установить имя пользователя и адрес email как значение для конкретного репозитория, то флаг `--global` не указывается (но перед выполнением такой команды вы должны непременно перейти в рабочий каталог активного репозитория, иначе Git выведет сообщение об ошибке). При наличии флага `--global` команда `git config` изменяет файл *.gitconfig* в вашем домашнем каталоге. Если флаг `--global` отсутствует, команда `git config` вносит изменения в файл *.git/config* для текущего репозитория.

Выполнение коммита изменений

После создания правильной конфигурации учетных записей пользователей система Git готова к выполнению коммита (commit) изменений, которые внесены в файлы, хранящиеся в репозитории. Напомним, что перед выполнением коммита изменений в репозиторий обязательно необходимо сначала внести в индекс (stage) соответствующие файлы с помощью команды `git add`. Эта операция выполняется как для новых, только что созданных файлов, так и для изменяемых файлов, которые уже хранятся в репозитории (этот вариант мы рассмотрим немного позже). Поскольку изменяемые файлы уже внесены в индекс (команда `git add` была выполнена в предыдущем разделе) и их состояние проверено (с помощью команды `git status`, которая сообщила, что файлы внесены в индекс (staged)), все готово к выполнению коммита.

Коммит изменений в репозиторий выполняется весьма просто – с помощью команды `git commit`:

```
vagrant@jessie:~/net-auto$ git commit -m "First commit to new repository"
[master (root-commit) 9547063] First commit to new repository
 3 files changed, 24 insertions(+)
 create mode 100644 sw1.txt
 create mode 100644 sw2.txt
 create mode 100644 sw3.txt
vagrant@jessie:~/net-auto$
```

Tlgm: @it_boooks



Если в команде `git commit` не указан флаг `-m`, то система Git запустит текстовый редактор, заданный по умолчанию, чтобы пользователь мог создать сообщение с описанием коммита. Текстовый редактор, вызываемый Git по умолчанию, можно переопределить в конфигурации (с помощью команды `git config` или отредактировав файл .gitconfig в домашнем каталоге). Например, можно сконфигурировать систему Git так, чтобы она запускала Sublime Text (https://www.sublimetext.com/), Atom (https://atom.io/) или любой другой текстовый редактор с графическим интерфейсом.

Теперь разберемся, что происходит в процессе коммита изменений в репозиторий. При добавлении файлов командой `git add` объекты, представляющие файлы (и их содержимое), были внесены в базу данных объектов системы Git. Точнее, Git создает блобы (blob – binary large object) для представления содержимого файлов и дерево объектов (tree objects) для представления структуры файлов и каталогов. При выполнении коммита изменений командой `git commit` в базу данных Git добавляется другой тип объекта – объект коммит (commit object), который ссылается на дерево объектов, которое, в свою очередь, ссылается на блобы. После создания объекта коммит наконец-то появляется текущая (зафиксированная по времени) ссылка на состояние всего репозитория в целом.

В настоящее время рассматриваемый в нашем примере репозиторий содержит единственный коммит, который можно просмотреть с помощью команды `git log`:

```
vagrant@jessie:~/net-auto$ git log
commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@jessie:~/net-auto$
```

Команда `git log` выводит различные коммиты или контрольные точки (checkpoints), созданные на протяжении всего жизненного цикла репозитория. При каждом коммите изменений создается новый объект коммит, который ссылается на общее состояние репозитория в момент создания этого коммита. Следовательно, можно наблюдать состояние репозитория и его содержимого в момент времени, соответствующий выполнению данного коммита. Таким образом, коммиты становятся «контрольными точками» (checkpoints), по которым можно перемещаться назад или вперед по хронологии репозитория.

Рекомендации по выполнению коммитов изменений

Понимание сущности процесса коммитов позволяет сформулировать несколько рекомендаций, относящихся к выполнению коммитов изменений в репозиторий:

- выполнение коммитов как можно чаще – можно просматривать состояние репозитория только в те моменты времени, когда выполнялись коммиты изменений. Если вы вносите изменения, сохраняете файлы, потом вносите новую порцию изменений и только после этого выполняете

Tlgm: @it_boooks

коммит, то вы не сможете просмотреть состояние репозитория на момент внесения первого набора изменений (потому что не был выполнен коммит);

- выполнение коммитов в логически обоснованные моменты времени – не следует выполнять коммит каждый раз после сохранения изменений в файле, хранящемся в репозитории. Может показаться, что это противоречит предыдущему пункту, но в действительности выполнение коммита имеет смысл только после завершения всего комплекса логически связанных изменений. Например, после обновления половины конфигурации коммутатора не имеет смысла выполнять коммит незавершенного набора изменений, поскольку вряд ли вы пожелаете откатиться к конфигурации коммутатора, находящейся в состоянии полуготовности. После завершения обновления конфигурации в полной мере необходимо выполнить коммит;
- использование сообщений с содержательным описанием коммитов – в выводе команды `git log` из предыдущего примера можно видеть, что сообщения коммита помогают понять, какие именно изменения содержались в этом коммите. Следует всегда стремиться к тому, чтобы сообщения коммита были полезными и понятными. Полгода спустя сообщение коммита может стать единственным способом, который поможет вспомнить, что именно вы изменили при выполнении этого коммита.

Прежде чем перейти к следующему разделу, необходимо обсудить еще одну тему. Выше было отмечено, что объекты в репозитории Git являются неизменяемыми, а также что изменения в объекте (например, в файле) приводят к созданию нового объекта (с соответствующей ссылкой по SHA-хеш-значению на содержимое этого объекта). Это утверждение верно для всех объектов репозитория Git, в том числе для блобов (содержимого файлов), деревьев объектов и объектов коммитов.

Но что делать, если после выполнения коммита вы обнаружили, что он содержал ошибку? Возможно, вы допустили пару опечаток в сетевой конфигурации или сообщение коммита было сформулировано неправильно. В этом случае Git позволяет изменить (или скорректировать – amend) самый последний коммит.

Корректировка коммитов

В том случае, когда самый последний коммит по какой-либо причине оказался ошибочным, существует возможность скорректировать (amend) этот коммит, воспользовавшись флагом `--amend` в команде `git commit`. Отметим, что вместо корректировки последнего коммита можно создать другой коммит, исправляющий ошибки предыдущего. Оба способа допустимы, каждый обладает собственными достоинствами и недостатками, которые мы рассмотрим в дальнейшем. Но сейчас необходимо продемонстрировать, как можно скорректировать коммит.

Tlgm: @it_boooks

Для корректировки самого последнего коммита нужно выполнить практически ту же самую последовательность действий, как и при выполнении обычного коммита:

1. Внести все необходимые изменения.
2. Зафиксировать (stage) изменения в индексе.
3. Выполнить команду `git commit --amend` для коммита зафиксированных изменений, определяя их как корректировку.

В действительности незаметно для пользователя Git создает новые объекты, поскольку это соответствует общим принципам функционирования Git и методике применения неизменяемых объектов, адресуемых по содержимому, но в хронологии репозитория пользователь видит только скорректированный коммит, а не первоначальную версию. Это позволяет поддерживать «более чистую» хронологию, хотя некоторые ревностные защитники философии Git утверждают, что наилучшей методикой остается простое выполнение нового корректирующего коммита (вместо использования флага `--amend`).

Какая методика лучше, определяет сам пользователь, но при выборе необходимо учесть несколько соображений. Если ведется совместная работа группы сотрудников в распределенной системе Git и репозиторий используется совместно, то применение флага `--amend` для корректировки коммитов, уже переданных в совместно используемый репозиторий, вообще говоря, является неудачным решением. Единственным исключением может стать вариант работы в среде, использующей Gerrit, в которой широко применяются корректирующие коммиты. Более подробно система Gerrit рассматривается в главе 10, а совместная работа группы сотрудников с Git описана в разделе «Совместная работа группы сотрудников с системой Git» текущей главы.

Внесение изменений и выполнение коммитов в отслеживаемые файлы

В предыдущих разделах мы создали репозиторий, добавили новые файлы и выполнили коммит изменений в репозиторий. Теперь необходимо рассмотреть, как вносятся изменения в файлы, которые уже хранятся в репозитории.

Процесс выполнения коммитов измененных версий файлов в репозиторий почти идентичен процессу выполнения коммитов, который был показан в предыдущем разделе:

1. Изменение файла (или нескольких файлов) в рабочем каталоге.
2. Фиксация внесенных изменений в индексе с помощью команды `git add`. Эта операция позволяет синхронизировать индекс с рабочим каталогом.
3. Выполнение коммита внесенных изменений с помощью команды `git commit`. Эта операция позволяет синхронизировать репозиторий с индексом и создает зафиксированную в текущий момент ссылку на состояние репозитория.

Рассмотрим этот процесс более подробно. Предположим, что необходимо отредактировать один из файлов, например *sw1.txt*, потому что изменилась

Tlgm: @it_boooks

конфигурация соответствующего коммутатора (или, возможно, потому что вы решили, что конфигурации должны развертываться только после регистрации их в системе управления исходным кодом). После изменения отслеживаемого файла (то есть файла, уже известного системе Git, которая отслеживает его состояние) команда `git status` показывает, что изменения действительно были внесены:

```
vagrant@trusty:~/net-auto$ git status
On branch master
(В главной ветви)
Changes not staged for commit:
(Изменения не зафиксированы для коммита)
  (use "git add <file>..." to update what will be committed)
  (используйте команду "git add <file>...", чтобы обновить данные, предназначенные для коммита)
  (use "git checkout -- <file>..." to discard changes in working directory)
  (используйте команду "git checkout -- <file>..." для отмены изменений в рабочем каталоге)

    modified:   sw1.txt

no changes added to commit (use "git add" and/or "git commit -a")
(нет изменений, добавленных в коммит (используйте команду "git add" и/или "git commit -a"))
vagrant@trusty:~/net-auto$
```

Отметим различие между сообщением о состоянии в этом примере и сообщением о состоянии в примере из предыдущего раздела. Система Git знает о существовании файла *sw1.txt* (он уже был добавлен в репозиторий), поэтому выводится другое сообщение о состоянии. Посмотрим, как изменится сообщение о состоянии, когда в рабочий каталог добавляется новый файл конфигурации коммутатора *sw4.txt*:

```
vagrant@trusty:~/net-auto$ git status
On branch master
(В главной ветви)
Changes not staged for commit:
(Изменения не зафиксированы для коммита)
  (use "git add <file>..." to update what will be committed)
  (используйте команду "git add <file>...", чтобы обновить данные, предназначенные для коммита)
  (use "git checkout -- <file>..." to discard changes in working directory)
  (используйте команду "git checkout -- <file>..." для отмены изменений в рабочем каталоге)

    modified:   sw1.txt

Untracked files:
(Неотслеживаемые файлы:)
  (use "git add <file>..." to include in what will be committed)
  (используйте команду "git add <file>..." для фиксации данных, предназначенных для коммита)

    sw4.txt

no changes added to commit (use "git add" and/or "git commit -a")
(нет изменений, добавленных в коммит (используйте команду "git add" и/или "git commit -a"))
vagrant@trusty:~/net-auto$
```

Tlgm: @it_boooks

И в этом случае система Git демонстрирует явное различие между отслеживаемыми изменениями в уже известном файле и обнаружением неотслеживаемых (не добавленных ранее) файлов в рабочем каталоге. Но в обоих случаях процесс внесения изменений (в отредактированном файле и в новом файле) в репозиторий абсолютно одинаков, и это можно наблюдать в выходных данных команды `git status`: рекомендуется сначала выполнить команду `git add`, затем команду `git commit`.

```
vagrant@trusty:~/net-auto$ git add sw1.txt
vagrant@trusty:~/net-auto$ git add sw4.txt
vagrant@trusty:~/net-auto$ git status
On branch master
(В главной ветви)
Changes to be committed:
(Изменения, для которых необходимо выполнить коммит:)
  (use "git reset HEAD <file>..." to unstage)
  (используйте команду "git reset HEAD <file>..." для отмены фиксации в индексе)

    modified:   sw1.txt
    new file:   sw4.txt

vagrant@trusty:~/net-auto$ git commit -m "Update sw1, add sw4"
[master 679c41c] Update sw1, add sw4
 2 files changed, 9 insertions(+)
 create mode 100644 sw4.txt
vagrant@trusty:~/net-auto$
```

Возможно, вы обратили внимание на то, что в выводе команды `git status` содержится ссылка на команду `git commit -a`. Флаг `-a` просто сообщает Git о необходимости добавления всех изменений из всех известных системе файлов. Если вы выполняете только коммит изменений в известные системе файлы и правильно подготовили все эти изменения в объединенном коммите, то использование команды `git commit -a` позволит обойтись без предварительной команды `git add`.

Но если необходимо распределить изменения в нескольких файлах по отдельным коммитам, то сначала потребуется выполнение команды `git add`, затем – `git commit`. При выборе конкретного подхода нужно принимать во внимание следующие обоснования:

- необходимость ограничения области видимости (действия) изменений одним коммитом, так как он оказывает меньшее воздействие на возможность возврата к более ранней версии;
- необходимость ограничения области видимости (действия) изменений одним коммитом, чтобы упростить для других сотрудников обзор сделанных вами изменений (более подробно это будет обсуждаться в главе 10);
- при совместной работе группы сотрудников «наилучшим практическим подходом» часто считается ограничение, определяющее включение в коммит только логически единого изменения, то есть можно включить в коммит только логически связанные друг с другом изменения, но не

Tlgm: @it_boooks

все изменения подряд. Некоторые общие правила и принципы групповой работы с Git будут рассматриваться ниже в разделе «Совместная работа группы сотрудников с системой Git».

Также следует отметить использование команды `git commit -m` в приведенном выше примере. Флаг `-m` позволяет пользователю включить сообщение коммита непосредственно в командную строку. Если флаг `-m` не указан, Git откроет текстовый редактор, определенный по умолчанию, чтобы пользователь получил возможность сформировать сообщение коммита. Сообщения коммита обязательны, и выше уже рекомендовалось включать в эти сообщения максимально возможный объем полезной информации. (В дальнейшем вы сами будете чрезвычайно благодарны за информативные сообщения коммитов при просмотре вывода команды `git log`.) Кроме того, можно объединять флаги `-a` и `-m`, например `git commit -am "Committing all changes to tracked files"`.



Чтобы получить более подоробное описание разнообразных флагов любой команды `git`, воспользуйтесь системой помощи `git help <command>`, например `git help commit` или `git help add`. При этом выводится страница оперативного руководства man, которая является частью документации Git. Если вы предпочитаете пользоваться непосредственно командой `man`, документация также доступна, но при этом необходимо объединять команды `git` с помощью дефиса. Например, чтобы вывести страницу man для команды `git commit`, нужно выполнить команду `man git-commit`.

После выполнения коммита другого набора изменений в репозиторий рассмотрим вывод команды `git log`:

```
vagrant@trusty:~/net-auto$ git log
commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000

    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@trusty:~/net-auto$
```

Теперь репозиторий содержит два коммита. Перед изучением возможностей просмотра репозитория в заданный момент времени (в момент конкретного коммита) сначала рассмотрим некоторые другие команды и выполним несколько дополнительных коммитов в репозиторий.

Отмена фиксации файлов в индексе

Если вы выполняли все команды примеров, то сейчас в репозитории содержатся четыре файла конфигурации коммутаторов (*sw1.txt–sw4.txt*) и два коммита. Допустим, что необходимо добавить пятый файл конфигурации коммутатора (разумеется, с именем *sw5.txt*). Процесс вам уже известен:

Tlgm: @it_boooks

1. Скопировать файл *sw5.txt* в рабочий каталог.
2. Использовать команду `git add` для фиксации файла из рабочего каталога в индексе.

В этот момент при выполнении команды `git status` выводится информация о том, что файл *sw5.txt* зафиксирован в индексе и готов к коммиту в репозиторий:

```
vagrant@jessie:~/net-auto$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   sw5.txt

vagrant@jessie:~$
```

Но после фиксации файла в индексе вы обнаружили, что в текущем состоянии содержащаяся в этом файле конфигурация не готова к коммиту в репозиторий. Возможно, конфигурация не полна, может быть, содержимое файла не точно соответствует действительной конфигурации устройства sw5 в сети. В подобной ситуации оптимальным решением будет отмена фиксации (unstage) этого файла в индексе.

Команда отмены фиксации файла в индексе, то есть удаление его из индекса, после чего рабочий каталог и индекс больше не являются синхронизированными, также доступна в системе Git. Если взглянуть на вывод команды `git status`, приведенной выше, то можно заметить, что Git сообщает способ отмены фиксации файла в индексе, предлагая следующий синтаксис:

```
git reset HEAD file
```

Описание этой команды необходимо начать с объяснения, что представляет собой `HEAD`. `HEAD` – это указатель на самый последний выполненный коммит (или самый последний коммит, зарегистрированный и подтвержденный в рабочем каталоге, но этот аспект мы пока еще не рассматривали). Напомним, что при фиксации файла в индексе (командой `git add`) содержимое рабочего каталога передается в индекс. При выполнении коммита (командой `git commit`) создается ссылка на текущий момент времени, указывающая на это содержимое, то есть собственно коммит. Каждый раз при выполнении очередного коммита система Git обновляет указатель `HEAD` таким образом, чтобы он ссылался на самый последний коммит.



Указатель HEAD также играет важную роль при начале работы с несколькими ветвями системы Git. Работа с ветвями пока еще не обсуждалась (они рассматриваются ниже в разделе «Работа с ветвями в Git»), но при изучении этой темы мы продолжим разъяснять роль HEAD в соответствующем контексте.

Для наглядной демонстрации сущности указателя HEAD можно привести небольшой пример. Если вы точно выполняли все примеры, приведенные выше в текущей главе, то вполне можете воспользоваться следующими командами

Tlgm: @it_boooks

(но учтите, что контрольная сумма SHA, показанная здесь, может отличаться от контрольной суммы SHA, полученной в вашей системе).

Сначала воспользуемся командой `cat`, чтобы вывести содержимое *.git/HEAD*:

```
vagrant@jessie:~/net-auto$ cat .git/HEAD
ref: refs/heads/master
vagrant@jessie:~/net-auto$
```

Очевидно, что HEAD представляет собой указатель на файл *refs/heads/master*. Если снова выполнить команду `cat` для вывода содержимого этого файла, то получим следующий результат:

```
vagrant@jessie:~/net-auto$ cat .git/refs/heads/master
679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
vagrant@jessie:~/net-auto$
```

Содержимое файла *.git/refs/heads/master* – это контрольная сумма SHA. Если теперь выполнить команду `git log`, то можно будет сравнить выведенную выше контрольную сумму SHA с самым последним коммитом, соответствующим этому значению:

```
vagrant@jessie:~/net-auto$ git log
commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000

    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@jessie:~/net-auto$
```

Обратите внимание на то, что контрольная сумма SHA самого последнего коммита совпадает со значением указателя HEAD (который указывает на файл *refs/heads/master*). Это свидетельствует о том, что HEAD действительно является указателем на самый последний коммит. В следующих разделах текущей главы будет показано, каким образом HEAD может также содержать ссылки на ветви и как этот указатель изменяется при регистрации контента в рабочем каталоге.

Но вернемся к команде `git reset`. Это команда с мощными возможностями, но, к счастью, для ее выполнения по умолчанию установлены некоторые безопасные параметры и опции. При самом простом варианте выполнения, то есть без флагов и без указания имени файла или пути к файлу, `git reset` всего лишь выводит индекс в виде содержимого, на которое ссылается указатель HEAD (как нам теперь известно, этот указатель ссылается на конкретный коммит, по умолчанию на самый последний коммит).

Напомним, что `git add` синхронизирует индекс с рабочим каталогом, выполняя операцию фиксации файла в индексе. Команда `git reset HEAD` *`file`* выполняет

Tlgm: @it_boooks

абсолютно противоположную операцию, в результате которой индекс выглядит как содержимое, на которое указывает HEAD. Эта команда отменяет (undo) изменения в индексе, сделанные командой `git add`, то есть отменяет фиксацию указанных файлов в индексе.

Следующий пример демонстрирует практическое применение команды `git reset`. Ранее файл *sw5.txt* был зафиксирован в индексе при подготовке выполнения его коммита в репозиторий, поэтому команда git status показывает этот файл в разделе «Изменения, для которых необходимо выполнить коммит:» (Changes to be committed:). Выполним команду `git reset` и посмотрим, как изменилось состояние репозитория:

```
vagrant@trusty:~/net-auto$ git reset HEAD sw5.txt
vagrant@trusty:~/net-auto$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    sw5.txt

nothing added to commit but untracked files present (use "git add" to track)
(в коммит ничего не добавлено, но имеются неотслеживаемые файлы (используйте команду "git
add", чтобы сделать их отслеживаемыми))
vagrant@trusty:~/net-auto$
```

Теперь файл *sw5.txt* уже не входит в список изменений, для которых необходимо выполнить коммит, вместо этого он показан как неотслеживаемый файл (то есть в индексе он не зафиксирован). После этого можно продолжить работу с содержимым файла *sw5.txt*, а когда файл конфигурации будет окончательно сформирован, выполнить коммит корректной версии в репозиторий.

В этом разделе было показано, как создать репозиторий, добавить в него файлы (новые и уже существующие), выполнить коммит изменений и отменить фиксацию файлов в индексе. А что делать с файлами, которые необходимо разместить вместе с другими файлами в репозитории, но они не должны отслеживаться системой Git? В этом случае поможет функция исключения файлов.

Исключение файлов из репозитория

Возможны ситуации, в которых некоторые файлы необходимо хранить в рабочем каталоге – «стартовой площадке» для репозитория Git, – но они не должны включаться в репозиторий. К счастью, Git предоставляет способ исключения конкретных файлов или имен файлов по шаблону из содержимого репозитория.

Вернемся к нашему примеру, в котором создан репозиторий, хранящий артефакты (объекты) автоматизации сети. Предположим, что в комплекте этих артефактов имеется скрипт на языке Python, который обеспечивает соединение с сетевыми коммутаторами для сбора информации от этих устройств. Ниже приведен пример такого скрипта, в нашем случае скрипт написан для установления соединения с коммутатором Arista и сбора информации:

Tlgm: @it_boooks


```
#!/usr/bin/env python

from pyeapi.client import Node as EOS
from pyeapi import connect
import yaml

def main():
    creds = yaml.load(open('credentials.yml'))

    un = creds['username']
    pwd = creds['password']

    conn = connect(host='eos-npab', username=un, password=pwd)
    device = EOS(conn)

    output = device.enable('show version')
    result = output[0]['result']
    print('Arista Switch Summary:')
    print('----------------------')
    print('OS Version:' + result['version'])
    print('Model:' + result['modelName'])
    print('System MAC:' + result['systemMacAddress'])

if __name__ == "__main__":
    main()
```

Часть этого скрипта функционирует с использованием данных аутентификации, хранящихся в отдельном файле, в нашем примере это YAML-файл *credentials.yml*. Данные аутентификации необходимо хранить вместе со скриптом на языке Python, но эти данные не должны отслеживаться и управляться репозиторием.



Включать или исключать секретные данные

Принятие решения о включении в репозиторий Git секретной информации, такой как пароли, SSH-ключи и т. п., полностью зависит от того, как будет использоваться репозиторий. В абсолютно частном репозитории, где не требуется хранение секретной информации отдельно для каждого пользователя, хранение такой информации, возможно, является удачным решением. Для репозиториев, где должна использоваться индивидуальная для каждого пользователя секретная информация, а также для совместно используемых репозиториев, открытых для доступа в той или иной степени, вероятнее всего, следует исключить секретную информацию из репозитория с помощью механизмов, рассматриваемых в текущем разделе.

К счастью, Git предоставляет способы исключения файлов, которые не должны отслеживаться как часть репозитория. Выше в текущей главе в разделе «Выполнение коммита изменений в репозиторий» обсуждались подробности конфигурации Git, характерные для репозитория, для пользователя и для всей системы в целом. Для механизма исключения файлов из репозитория Git принят подобный подход, то есть существуют способы исключения файлов из конкретного репозитория или по принадлежности конкретному пользователю.

Tlgm: @it_boooks

Исключение файлов из конкретного репозитория

Начнем с методики исключения файлов из заданного репозитория. Чаще всего применяется способ исключения (или игнорирования) файлов с использованием специального файла *.gitignore*, хранящегося в самом репозитории. Как и любое другое содержимое репозитория, файл *.gitignore* обязательно должен быть зафиксирован в индексе, и при любых изменениях этого файла выполняются коммиты в репозиторий. Преимущество такого подхода состоит в том, что в дальнейшем файл *.gitignore* распространяется как часть репозитория. Это очень удобно при работе группы, все члены которой используют Git как распределенную систему управления версиями (DVCS – distributed version control system).

Содержимое файла *.gitignore* – это простой список имен файлов или шаблонов имен файлов, размещаемых по одному в строке. Чтобы сформировать собственный список файлов, игнорируемых системой Git, нужно создать файл *.gitignore* в рабочем каталоге, добавить в него имена или шаблоны имен исключаемых файлов, зафиксировать этот файл в индексе и выполнить коммит в репозиторий.

Возвращаясь к скрипту на языке Python из предыдущего раздела, вспомним, что в нем используется файл с регистрационными данными *credentials.yml*. Чтобы система Git не отслеживала изменения в этом файле, создадим файл *.gitignore* (если он не был создан ранее):

1. Создать пустой файл командой `touch .gitignore`.
2. Отредактировать файл *.gitignore* с помощью текстового редактора по вашему выбору: добавить имя файла *credentials.yml* в отдельной строке.

Если в этот момент выполнить команду `git status`, то можно увидеть, что система Git заметила добавление файла *.gitignore*, но файл *credentials.yml* не попал в список кандидатов на включение в репозиторий:

```
vagrant@trusty:~/net-auto$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
vagrant@trusty:~/net-auto$
```

Теперь можно зафиксировать в индексе и выполнить коммит файла *.gitignore* в репозиторий с помощью команды `git commit -am "Adding .gitignore file"`.

Если после этого будет создан файл *credentials.yml* для Python-скрипта, то система Git не обратит на него никакого внимания. В приведенном ниже листинге можно видеть, что этот файл существует в рабочем каталоге, но команда `git status` не выводит никаких сообщений об изменениях или о неотслеживаемых файлах.

```
[vagrant@centos net-auto]$ ls -la
total 40
drwxrwxr-x 3 vagrant vagrant 4096 May 31 16:32 .
```

Tlgm: @it_boooks


```
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 May 31 16:34 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 credentials.yml
-rwxrwxr-x 1 vagrant vagrant    0 May 31 16:32 script.py
-rw-rw-r-- 1 vagrant vagrant   98 May 12 20:22 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
[vagrant@centos net-auto]$ git status
On branch master
nothing to commit, working directory clean
[vagrant@centos net-auto]$
```

Внимательный читатель может отметить: тот факт, что система Git не сообщает о необходимости коммита, еще не означает, что файл полностью игнорируется. Рассмотрим еще несколько команд, которые помогут проверить действительное состояние. Сначала воспользуемся командой `git log` для вывода хронологии коммитов:

```
vagrant@jessie:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@jessie:~/net-auto$
```

Теперь выполним запрос к системе Git, чтобы увидеть версии содержимого репозитория в каждый из перечисленных выше моментов времени. Для этого используем команду `git ls-tree` с указанием хеш-значения SHA проверяемого коммита. Возможно, вы уже заметили, что Git часто использует только первые семь символов хеш-значения SHA, как, например, в приведенном выше выводе команды `git log --oneline` (при необходимости Git автоматически увеличивает количество выводимых символов, чтобы хеш-значения оставались различными). Почти в каждом случае (исключения крайне редки) такой подход применим для команд, требующих ввода хеш-значения SHA. Например, чтобы увидеть состояние репозитория на момент предпоследнего коммита (хеш-значение SHA которого начинается с символов `5cd13a8`), необходимо выполнить следующую команду:

```
vagrant@trusty:~/net-auto$ git ls-tree 5cd13a8
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    script.py
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@trusty:~/net-auto$
```

Tlgm: @it_boooks

Здесь можно видеть, что на момент выполнения этого коммита файл *credentials.yml* не присутствует в репозитории. Проверим состояние на момент самого последнего коммита.

```
vagrant@jessie:~/net-auto$ git ls-tree ed45c95
100644 blob 2c1817fdecc27ccb3f7bce3f6bbad1896c9737fc    .gitignore
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    script.py
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@jessie:~/net-auto$
```

(Предлагаем читателю в качестве самостоятельного упражнения выполнить проверки всех остальных коммитов, чтобы окончательно убедиться в том, что файл *credentials.yml* не включается в репозиторий после какого-либо коммита.)

Глобальное исключение файлов

В дополнение к методике исключения файлов из конкретного репозитория с применением файла *.gitignore* в рабочем каталоге этого репозитория можно также создать глобальный файл для исключения файлов из всех репозиториев на вашем компьютере. Для этого нужно создать файл *.gitignore_global* в домашнем каталоге и добавить в него имена исключаемых файлов. Также можно выполнить приведенную ниже команду для полной уверенности в том, что система Git правильно сконфигурирована для использования нового файла *.gitignore_global*, размещенного в домашнем каталоге:

```
git config --global core.excludesfile /path/to/.gitignore_global
```

Если файл *.gitignore_global* размещен в домашнем каталоге, то путь к нему обычно указывается в следующем виде: *~/.gitignore_global*.

Использование команд `git log` и `git ls-tree` вполне естественным образом подводит нас к следующей теме: получение более подробной информации о репозитории, о его хронологии и содержимом.

Получение более подробной информации о репозитории

Ранее при необходимости получения более подробной информации о репозитории уже были продемонстрированы некоторые возможности команды `git log`. Эта команда использовалась в нескольких примерах, тем самым подтверждая свою полезность.

Получение общей информации из журнала

В самой общей форме команда `git log` выводит хронологию коммитов вплоть до HEAD, поэтому выполнение команды `git log` позволит получить список всех коммитов, зарегистрированных в хронологии репозитория. Ниже приведен вывод этой команды для примера репозитория, рассматриваемого в текущей главе:

Tlgm: @it_boooks


```
[vagrant@centos net-auto]$ git log
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file

commit 5cd13a84de0e01f358636dee98da2df7d95e17ea
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:33:41 2016 +0000

    Add Python script to talk to network switches

commit 2a656c3288d5a324fba1c2cbfccbc0e29db73969
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 14:56:56 2016 +0000

    Add configuration for sw5

commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000

    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
[vagrant@centos net-auto]$
```

Получение краткой информации из журнала

Для команды `git log` имеется большое количество разнообразных флагов и опций, поэтому невозможно подробно рассмотреть все ее возможности в рамках текущего раздела. Ранее уже использовался один из наиболее часто применяемых полезных флагов `--oneline`, который позволяет получить показанный ниже краткий отчет о состоянии репозитория из нашего примера:

```
[vagrant@centos net-auto]$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$
```

Здесь можно видеть, что флаг `--oneline` сокращает хеш-значение SHA до семи символов и выводит только список сообщений коммитов. Для репозиториев с длинной хронологией, возможно, удобнее начинать просмотр с команды `git log --oneline`, а затем приступать к изучению отдельных коммитов.

Запрещение поведения системы Git, определенного по умолчанию, при котором выводимая информация по программному каналу (pipe) передается в программу постраничного (поэкранного) вывода (pager), предоставляет возможность выполнять поиск

Tlgm: @it_boooks

в выводимых данных с помощью утилиты grep. Для запрещения постраничного (по-экранного) вывода в системе Git используйте флаг `--no-pager`, например `git --no-pager log --oneline`.

Существует несколько различных флагов и опций для получения подробной информации о каждом конкретном коммите. Во-первых, можно в команде `git log` получить заданный набор (диапазон) коммитов. Синтаксис: `git log` *`start SHA..end SHA`*. Если необходимо вывести подробности о паре последних коммитов в репозиторий из нашего примера, то команда может выглядеть следующим образом (если вы задумались о том, откуда взяты заданные хеш-значения SHA, то внимательно посмотрите на вывод команды `git log --oneline` из предыдущего примера в текущем разделе и вспомните, что достаточно указать первые семь символов хеш-значения SHA):

```
vagrant@trusty:~/net-auto$ git log 5cd13a8..ed45c95
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file
vagrant@trusty:~/net-auto$
```

В системе Git также существует несколько символических имен, которые можно использовать в различных командах, в том числе и в команде `git log`. Ранее уже применялось символическое имя HEAD. Если необходимо исследовать коммит, выполненный непосредственно перед коммитом, зафиксированным в HEAD, то можно применить указатель в символической форме `HEAD~1`. Чтобы получить данные о коммите, расположенном на две позиции раньше HEAD, используется символический указатель `HEAD~2`, на три позиции раньше – `HEAD~3` и т. д. (Вы можете собственноручно проверить работу этого шаблона на практике.) В нашем конкретном примере для конкретного рассматриваемого репозитория приведенная ниже команда выводит тот же результат, что и команда из предыдущего примера:

```
vagrant@trusty:~/net-auto$ git log HEAD~1..HEAD
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file
vagrant@trusty:~/net-auto$
```

Когда мы продолжим изучение указателя HEAD в следующих разделах, вы поймете, почему было особо отмечено, что «в нашем конкретном примере для конкретного рассматриваемого репозитория» две команды `git log` выводят абсолютно одинаковый результат.

Получение подробной информации о конкретных коммитах

Другим способом получения подробностей о конкретном коммите является использование команды `git cat-file`. Git, как и многие другие инструменталь-

Tlgm: @it_boooks

ные средства Unix/Linux, интерпретирует любую сущность (объект) как файл. Следовательно, коммит также можно интерпретировать как файл, «содержимое» которого выводится на экран. Именно эту операцию выполняет команда `git cat-file`. Таким образом, указав сокращенное хеш-значение SHA для конкретного коммита, можно увидеть более подробную информацию об этом коммите. Результат выполнения приведен ниже:

```
vagrant@jessie:~/net-auto$ git cat-file -p 2a656c3
tree 9f955969460fe47cb3b22d44e497c7a76c7a8db2
parent 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
author John Smith <john.smith@networktocode.com> 1464706616 +0000
committer John Smith <john.smith@networktocode.com> 1464706616 +0000

Add configuration for sw5
vagrant@jessie:~$
```

(Флаг `-p` в команде `git cat-file` позволяет выполнить некоторое форматирование вывода на основе типа файла. Более подробно о действии этого и других флагов можно узнать из man-страницы команды `git cat-file`.)

Возможно, вы заметили, что в приведенном выше выводе содержатся некоторые элементы информации, которые не показывает по умолчанию команда `git log`: хеш-значение SHA родительского коммита и хеш-значение SHA дерева объектов. Первое значение можно использовать для просмотра коммита, который является «родительским» по отношению к текущему. У каждого коммита есть родительский коммит или коммит-предок, который позволяет отследить всю цепочку коммитов вплоть до самого первого, который является единственным коммитом в репозитории, не имеющим предка. Хеш-значение SHA дерева объектов фиксирует файлы, находящиеся в репозитории на момент выполнения рассматриваемого коммита. Ранее мы уже использовали это значение в команде `git ls-tree`, как показано ниже:

```
vagrant@jessie:~/net-auto$ git ls-tree 9f9559
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@jessie:~/net-auto$
```

Воспользовавшись контрольными суммами SHA, приведенными в этом примере, можно выполнять команду `git cat-file` для просмотра содержимого каждого из перечисленных файлов в определенный момент времени (это время выполнения конкретного коммита).

Рассмотрим более подробно, как это работает. В следующем примере сначала используется команда `git log --oneline` для вывода хронологии коммитов в репозиторий. Затем выполняются команды `git cat-file` и `git ls-tree` с соответствующими семисимвольными хеш-значениями SHA для вывода содержимого конкретно указанного файла в два различных момента времени (в моменты выполнения двух различных коммитов).

Tlgm: @it_boooks


```
vagrant@trusty:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ git cat-file -p 9547063
tree fdad0ff90745deb944a430e2151d085aebc68d00
author John Smith <john.smith@networktocode.com> 1463074642 +0000
committer John Smith <john.smith@networktocode.com> 1463074642 +0000

First commit to new repository
vagrant@trusty:~/net-auto$ git ls-tree fdad0f
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
vagrant@trusty:~/net-auto$ git cat-file -p 02df3d
interface ethernet0

interface ethernet1

interface ethernet2

interface ethernet3

vagrant@trusty:~/net-auto$
```

Здесь показано содержимое файла *sw1.txt* как первоначального коммита. Теперь повторим тот же процесс для второго коммита:

```
vagrant@trusty:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ git cat-file -p 679c41c
tree a093d5f26677d345cb274ceb826e70bdb31ffd6f
parent 95470631aba32d6823c80fdd3c6f923824dde470
author John Smith <john.smith@networktocode.com> 1463085679 +0000
committer John Smith <john.smith@networktocode.com> 1463085679 +0000

Update sw1, add sw4
vagrant@trusty:~/net-auto$ git ls-tree a093d5
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
vagrant@trusty:~/net-auto$ git cat-file -p 2567e0
interface ethernet0
  duplex auto

interface ethernet1

interface ethernet2

interface ethernet3

vagrant@trusty:~/net-auto$
```

Tlgm: @it_boooks

Обратите внимание на изменение в содержимом файла *sw1.txt*. Но выполненные в приведенных выше примерах операции в какой-то мере избыточны – должен существовать более простой способ определения различий между двумя версиями файла в репозитории. Поэтому в следующем разделе мы рассмотрим команду `git diff`.

Определение различий между версиями файлов

В начале текущей главы было отмечено одно из преимуществ использования системы управления версиями для артефактов автоматизации сети (конфигураций коммутаторов, скриптов на языке Python, шаблонов на языке Jinja и т. д.) – возможность определения различий между версиями файлов, изменяемых время от времени. В предыдущем разделе демонстрировалась методика определения таких различий «вручную» (с помощью последовательности команд). Здесь мы рассмотрим более простой способ: выполнение команды `git diff`.



Следует отметить, что система Git также поддерживает совместную работу с инструментальными средствами определения различий в версиях diff, в том числе с diff-инструментами, предоставляющими графический пользовательский интерфейс. В таких случаях вместо `git diff` используется команда `git difftool`.

Определение различий между коммитами

Команда `git diff` показывает различия между версиями файла (то есть отличие содержимого файла в некоторый момент времени от содержимого того же файла в другой момент времени). Необходимо лишь указать два коммита и проверяемый файл. Ниже приведен пример, в котором сначала выводится хронология коммитов с помощью команды `git log`, затем выполняется команда `git diff` для сравнения двух версий указанного файла.

```
[vagrant@centos net-auto]$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$ git diff 9547063..679c41c sw1.txt
diff --git a/sw1.txt b/sw1.txt
index 02df3d4..2567e07 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,4 +1,5 @@
 interface ethernet0
+  duplex auto

 interface ethernet1

[vagrant@centos net-auto]$
```

На первый взгляд формат, в котором команда `git diff` выводит обнаруженные различия между версиями файлов, может показаться несколько стран-

Tlgm: @it_boooks

ным. Здесь необходимо понять смысл строк, выводимых непосредственно после строки `index 02df3d4..2567e07 100644`. С помощью дефисов команда `git diff` обозначает версию файла *a* (`--- a/sw1.txt`), а с помощью символов плюс – версию файла *b* (`+++ b/sw1.txt`). Далее при определении различий в версиях строки, существующие в версии *a*, обозначаются префиксом в виде дефиса, а для строк из версии *b* предназначен префикс в виде символа плюс. Если строки одинаковы в обеих версиях, то перед ними размещается пробел.

Таким образом, в приведенном выше примере можно видеть, что при выполнении самого последнего коммита, представленного хеш-значением `679c41c`, была добавлена строка `duplex auto`. Разумеется, это самый простой пример, но мы надеемся, что вы уже оценили полезность команды `git diff`.



Пропуск имен файлов в командах

Если вы пропустили имя файла в команде `git diff` (например, введена команда `git diff start SHA..end SHA`), то система Git покажет различия для всех файлов, измененных в процессе выполнения этого коммита, вместо различий в версиях одного файла, заданного в командной строке. Добавление имени файла в команду `git diff` позволяет сосредоточить внимание на изменениях единственного конкретного файла.

Просмотр других типов различий

Внесем несколько изменений в файл конфигурации *sw1.txt*, чтобы задача определения различий между версиями стала несколько более сложной. Кроме того, продемонстрируем некоторые другие возможные способы использования команды `git diff`.

С помощью текстового редактора по вашему выбору внесите несколько любых изменений в файл *sw1.txt* (в данном случае абсолютно не важно, какие именно изменения будут внесены). Потом выполняется команда `git status` для подтверждения факта существования внесенных изменений в рабочем каталоге. И еще раз перед фиксацией этих изменений в индексе выполним команду `git diff`.

```
vagrant@trusty:~/net-auto$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   sw1.txt

no changes added to commit (use "git add" and/or "git commit -a")
vagrant@trusty:~/net-auto$ git diff
diff --git a/sw1.txt b/sw1.txt
index 2567e07..7005dc6 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
```

Tlgm: @it_boooks


```
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
vagrant@trusty:~/net-auto$
```

При выполнении команды `git diff` в таком виде – без каких-либо параметров и опций – выводятся все различия между текущим рабочим деревом и индексом. Таким образом, здесь показаны все изменения, которые пока еще не зафиксированы с помощью следующего коммита.

Теперь зафиксируем эти изменения, чтобы подготовить следующий коммит, затем рассмотрим, как можно использовать команду `git diff` несколько другим способом:

```
vagrant@jessie:~/net-auto$ git add sw1.txt
vagrant@jessie:~/net-auto$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   sw1.txt

vagrant@jessie:~/net-auto$ git diff
vagrant@jessie:~/net-auto$ git diff --cached
diff --git a/sw1.txt b/sw1.txt
index 2567e07..f3b5ad5 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
vagrant@jessie:~/net-auto$
```

Можно видеть, что первая команда `git diff` не возвращает какой-либо результат, что вполне оправдано – нет никаких изменений, которые зафиксированы в индексе для следующего коммита. Но при добавлении ключа `--cached` команда `git diff` показывает различия между индексом и HEAD. Иначе говоря,

Tlgm: @it_boooks

последняя форма команды `git diff` показывает различия между индексом и самым последним коммитом.

После завершения выполнения коммита с самым последним набором изменений можно повторить весь цикл команд `git diff` в различных формах. Это позволит наблюдать изменения между любыми двумя коммитами.

```
[vagrant@centos net-auto]$ git commit -m "Defined VLANs on sw1"
[master 3588c31] Defined VLANs on sw1
 1 file changed, 4 insertions(+), 2 deletions(-)
[vagrant@centos net-auto]$ git status
On branch master
nothing to commit, working directory clean
[vagrant@centos net-auto]$ git log --oneline
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$ git diff 679c41c..3588c31 sw1.txt
diff --git a/sw1.txt b/sw1.txt
index 2567e07..f3b5ad5 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
[vagrant@centos net-auto]$
```

Прежде чем перейти к следующей теме – создание ветвей в системе Git, – подведем краткие итоги по изложенной выше информации. В текущем разделе подробно рассматривались следующие операции:

- фиксация изменений в индексе (команда `git add`) и выполнение их коммита в репозиторий (команда `git commit`);
- изменение конфигурации Git (команда `git config`);
- отмена фиксации изменений, которые пока еще не готовы к коммиту (команда `git reset`);
- исключение файлов из набора отслеживаемых в репозитории (использование файла *.gitignore*);
- просмотр хронологии репозитория (команда `git log`);

Tlgm: @it_boooks

- сравнение различных версий файлов в репозитории с целью обнаружения конкретных различий в каждой версии (команда `git diff`).

В следующем разделе будет подробно рассматриваться одна из наиболее мощных функциональных возможностей системы Git: создание ветвей версий.

Создание ветвей версий в системе Git

Если обратиться к разделу «Краткая история создания и развития Git» в начале текущей главы, то можно отметить, что одной из основных целей создания Git являлась полноценная поддержка процесса нелинейной разработки. Легко сказать, что в Git требуется поддержка деятельности нескольких разработчиков, одновременно работающих над одним и тем же проектом. Но как это сделать? В системе Git такой подход реализован через использование ветвей (branches).

В системе Git ветвь (branch) – это указатель на коммит. Сначала такое определение выглядит не очень внушительным, поэтому рассмотрим несколько графических схем, которые помогут лучше понять концепцию ветвей и четко обосновать, почему механизм поддержки нелинейной разработки в системе Git действительно обладает столь мощными функциональными возможностями.

Напомним, что в разделе «Обзор архитектуры Git» было отмечено использование в системе Git следующего набора объектов: блобы (представляющие содержимое файлов в репозитории), деревья (представляющие структуру файлов и каталогов в репозитории) и коммиты (представляющие «снимки» состояния репозитория в определенный момент времени, то есть структуру репозитория и его содержимое). Все перечисленные выше объекты графически изображены на рис. 8.1.

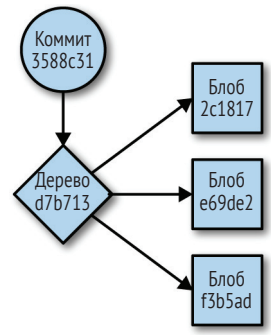


Рис. 8.1 ❖ Объекты в репозитории Git

Каждый из этих объектов идентифицируется по хеш-значению SHA-1, вычисленному по содержимому. В предыдущих разделах было показано, как осуществляется обращение к коммитам через соответствующее хеш-значение SHA-1 и как применяются команды `git ls-tree` и `git cat-file` для просмотра со-

Tlgm: @it_boooks

держимого таких объектов, как деревья и блобы соответственно также с помощью ссылок по их хеш-значению SHA-1.

При внесении изменений и выполнении их коммитов в репозиторий создаются новые коммит-объекты (увеличивается количество мгновенных «снимков»), каждый из которых указывает на предыдущий коммит (обозначаемый термином «родительский коммит», как было отмечено в разделе «Получение более подробной информации о репозитории» текущей главы). Состояние после выполнения нескольких коммитов можно графически изобразить так, как это сделано на рис. 8.2 (для упрощения схемы блобы не показаны).

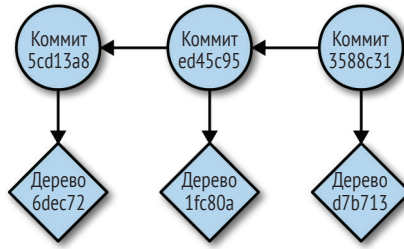


Рис. 8.2 ❖ Цепочка коммитов в репозитории Git

Каждый коммит указывает на объект-дерево, а каждый объект-дерево содержит указатели на блобы, представляющие содержимое репозитория на момент соответствующего коммита. Используя ссылку на объект-дерево и ссылки на связанные с ним блобы, можно воссоздать состояние репозитория на момент любого коммита. Таким образом, можно считать коммиты мгновенными снимками состояния репозитория.

Все сказанное выше помогает более полно описать архитектуру системы Git, но какое отношение это имеет к концепции создания ветвей в Git? Для ответа на этот вопрос (который будет дан очень скоро) необходимо вернуться к концепции указателя HEAD. В разделе «Отмена фиксации файлов в индексе» текущей главы HEAD определен как указатель на самый последний коммит или на коммит, зарегистрированный в рабочем каталоге. Графическое представление указателя HEAD можно изобразить так, как показано на рис. 8.3.

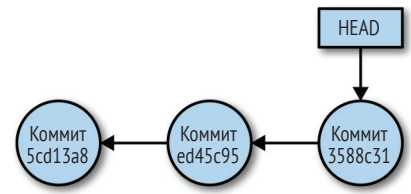


Рис. 8.3 ❖ Указатель HEAD,
ссылающийся на самый последний коммит

Tlgm: @it_boooks

Этот факт можно проверить с помощью процедуры, описанной ранее в текущей главе (предполагается, что вы не обращались к другой ветви или к другому коммиту – причина этого ограничения будет объяснена немного позже):

1. В рабочем каталоге репозитория выполнить команду `cat .git/refs/heads/master`. Запомнить выведенное значение.
2. Сравнить значение, выведенное после выполнения приведенной выше команды, со значением самого последнего коммита, которое выводится при выполнении команды `git log --oneline`. В обоих случаях вы должны увидеть совершенно одинаковые значения. Это подтверждает, что HEAD указывает на самый последний коммит.

По умолчанию каждый репозиторий Git начинает работу с единственной ветвью, которая называется основной (master). Как единственная ветвь, содержащая указатель на коммит, основная ветвь изображена на рис. 8.4.

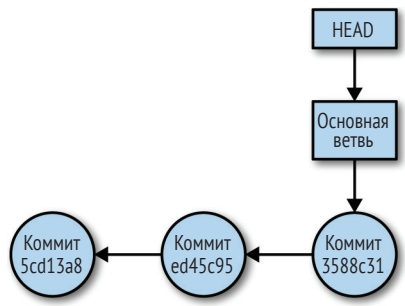


Рис. 8.4 ❖ HEAD указывает на самый последний коммит в основной ветви

Из схемы на рис. 8.4 видно, что основная ветвь указывает на коммит, а HEAD ссылается на указатель основной ветви.

Но пользователь не ограничен лишь одной ветвью в репозитории Git. В действительности концепция ветвей очень проста и очевидна (это всего лишь указатель на коммит), поэтому настоятельно рекомендуется использовать несколько ветвей. После создания новой ветви (назовем ее testing (ветвь для тестирования), хотя имя не имеет особого значения) структура объектов Git выглядит так, как показано на рис. 8.5.

Новая ветвь создана (команды для ее создания будут описаны несколько позже), и она указывает на конкретный коммит. Но указатель HEAD не изменился, потому что он меняется при обращении к какому-либо содержимому в репозитории. Поэтому для того, чтобы HEAD указывал на новую ветвь, необходимо к ней обратиться (check out). Это отчасти похоже на случай, когда нужно работать с репозиторием в некоторый более ранний момент времени (во время выполнения более раннего коммита), поэтому нужно обратиться к этому конкретному коммиту (то есть «извлечь» этот коммит). После обращения к новой ветви HEAD перемещает указатель на эту ветвь, как показано на рис. 8.6.

Tlgm: @it_boooks

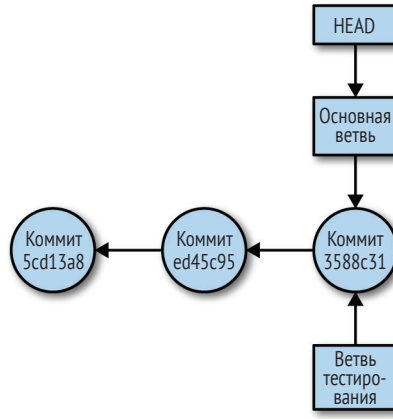


Рис. 8.5 ❖ Новая ветвь, созданная в репозитории Git

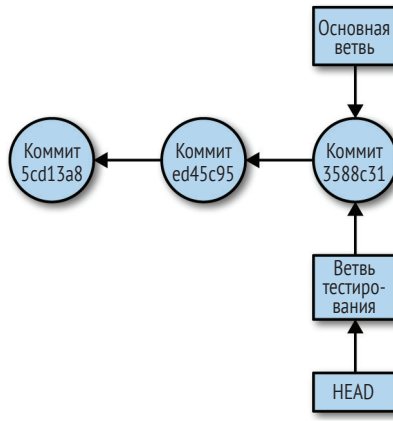


Рис. 8.6 ❖ HEAD указывает на ветвь, к которой происходило самое последнее обращение

В этот момент можно вносить изменения и выполнять их коммиты в репозиторий. Именно здесь в полной мере начинает проявляться вся мощь концепции ветвей: *новые изменения полностью изолируются от основной ветви*. Предположим, что вы внесли несколько изменений в ветвь testing и выполнили коммит этих изменений в репозиторий. Графическое представление объектов и отношений между ними внутри репозитория после выполненных операций показано на рис. 8.7.

Отметим, что ветвь testing и соответственно указатель HEAD переместились «вперед», чтобы отобразить самый последний коммит, но основная ветвь (master) при этом остается неизменной. В любой момент можно обратиться

Tlgm: @it_boooks

к основной ветви и вернуться к состоянию, существовавшему перед созданием новой ветви и внесением в нее изменений. Ниже приведена схема, демонстрирующая пример развития нескольких ветвей во времени, что позволяет вести разработку с внесением критических исправлений (hotfixes), с вводом новых функциональных возможностей и выпуском новых релизов без какого-либо влияния на основную ветвь.

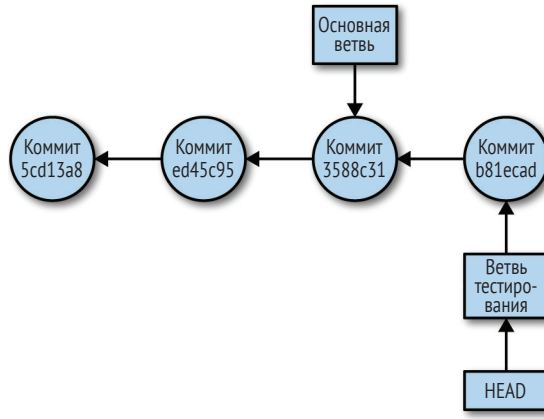


Рис. 8.7 ❖ Добавление коммита в ветвь, альтернативную основной

На рис. 8.8 показан пример более сложной структуры репозитория с несколькими ветвями, позволяющий получить представление о том, как можно использовать ветви в процессе обычной разработки программного обеспечения.

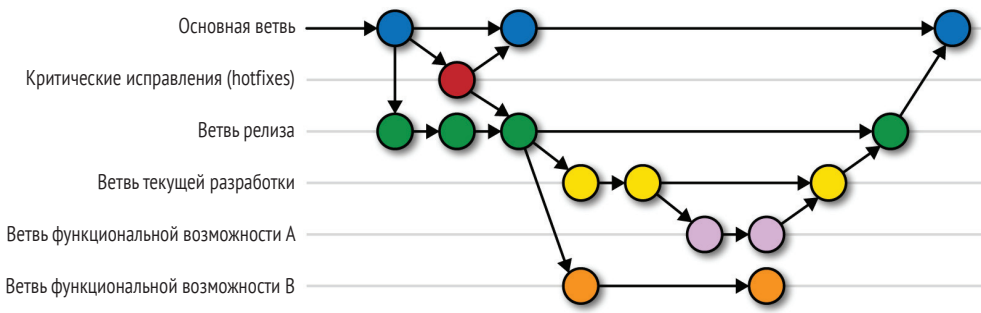


Рис. 8.8 ❖ Несколько ветвей репозитория в цикле разработки

Можно предложить еще некоторые возможности практического применения ветвей:

Tlgm: @it_boooks

- создание новой ветви, если необходимо попробовать применить что-то новое или совершенно отличающееся от существующего без какого-либо воздействия на основную ветвь. Если проверка оказалась неудачной, ничего страшного не происходит – основная ветвь осталась неизменной;
- ветви образуют основу для совместной работы нескольких разработчиков (авторов) в одном репозитории. Если вы работаете в ветви A, а ваш коллега – в ветви B, то можно с полной уверенностью утверждать, что вносимые изменения никоим образом не повлияют на соседнюю ветвь. (Разумеется, могут возникать вопросы и проблемы в тот момент, когда приходит время объединения (merge) изменений из двух ветвей в единое целое, но это совершенно другая тема, которая будет рассматриваться в разделе «Объединение и удаление ветвей» текущей главы.)

Теперь приступим к изучению непосредственной работы с ветвями Git, то есть проверим теорию практикой.

Создание ветви

Для создания ветви в системе Git (напомним, что это всего лишь указатель на коммит) используется команда `git branch`. Таким образом, для создания ветви testing, которая рассматривалась в предыдущем разделе, нужно просто выполнить команду `git branch testing`. Команда не выводит никакой информации, но существует способ проверки правильности ее выполнения.

Сначала можно проверить содержимое каталога *.git/refs/heads*, в котором должна появиться новая запись, соответствующая только что созданной ветви. При выполнении команды `cat` для вывода содержимого нового файла можно видеть, что в файле хранится указатель на тот же коммит, на который ссылается HEAD в момент создания новой ветви. Ниже приведены результаты выполнения описанных команд:

```
vagrant@trusty:~/net-auto$ git branch testing
vagrant@trusty:~/net-auto$ ls -la .git/refs/heads
total 16
drwxrwxr-x 2 vagrant vagrant 4096 Jun  4 19:16 .
drwxrwxr-x 4 vagrant vagrant 4096 May 12 14:52 ..
-rw-rw-r-- 1 vagrant vagrant   41 Jun  4 19:16 master
-rw-rw-r-- 1 vagrant vagrant   41 Jun  4 19:13 testing
vagrant@trusty:~/net-auto$ cat .git/refs/heads/testing
3588c31cbf958fe1a28d5e1f19ace669de99bb8c
vagrant@trusty:~/net-auto$ git log --oneline
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$
```

Tlgm: @it_boooks

Наличие файла *testing* в каталоге *.git/refs/heads* и содержимое этого файла, указывающее на самый последний коммит в момент создания ветви, подтверждает, что новая ветвь действительно была создана. Кроме того, факт создания можно проверить, выполнив команду `git branch` без указания имени. При этом выводится список всех существующих ветвей. Активная ветвь (active branch), то есть ветвь, выбранная или «извлеченная» (checked out) для использования в рабочем каталоге, помечена звездочкой перед ее именем (а также может быть выделена другим цветом, если терминал поддерживает эту функцию). При этом можно видеть, что новая ветвь testing была создана, но не выбрана (не «извлечена») – основная ветвь (master) остается активной.

Для изменения активной ветви необходимо выбрать или «извлечь» требуемую ветвь.

Выбор активной ветви

Выбор или «извлечение» (check out) ветви означает, что эта ветвь становится активной и будет доступна в рабочем каталоге для редактирования и внесения изменений. Для выбора активной ветви применяется команда `git checkout` с указанием имени требуемой ветви, которую необходимо сделать активной:

```
vagrant@jessie:~/net-auto$ git branch
* master
  testing
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$
```



Можно одновременно создать ветвь и сделать ее активной с помощью команды `git checkout -b` *<имя_ветви>*.

Внесем простое изменение в репозиторий, например добавим файл, а затем переключимся обратно в основную ветвь master, чтобы посмотреть, как система Git обработала внесенное изменение. Сначала зафиксируем файл *sw6.txt* в индексе репозитория и выполним соответствующий коммит в ветвь testing:

```
[vagrant@centos net-auto]$ git add sw6.txt
[vagrant@centos net-auto]$ git commit -m "Add sw6 configuration"
[testing b45a2b1] Add sw6 configuration
 1 file changed, 12 insertions(+)
 create mode 100644 sw6.txt
[vagrant@centos net-auto]$
```

Отметим, что ответ системы Git при коммите изменения включает имя ветви и хеш-значения SHA этого коммита (`[testing b45a2b1]`). Команда `git log --oneline` позволяет проверить совпадение хеш-значения самого последнего коммита и хеш-значения, выводимого командой `git commit`. Кроме того, команда `cat .git/HEAD` подтвердит, что вы действительно находитесь в ветви testing (потому что HEAD указывает на *.git/refs/heads/testing*), а команда `cat .git/refs/`

Tlgm: @it_boooks

heads/testing также покажет хеш-значение SHA самого последнего коммита. Все это подтверждает, что HEAD действительно указывает на самый последний коммит в ветви, которая была выбрана активной.

Получение информации о ветвях системы Git в обычной командной строке

При работе с несколькими ветвями в репозитории Git иногда трудно узнать, какая ветвь в текущий момент является активной (checked out). Для решения этой проблемы в большинство дистрибутивных пакетов Git версии 1.8 и более поздних включена поддержка взаимодействия с bash – командной оболочкой, которая по умолчанию используется во многих дистрибутивах Linux. Это позволяет вывести текущую активную ветвь Git в командной строке bash. В Ubuntu 14.04 и Debian 8.x файл поддержки bash называется git-sh-prompt и размещается в каталоге */usr/lib/git-core*. В CentOS 7 (и предположительно в RHEL 7) файл поддержки именуется git-prompt.sh и находится в каталоге */Library/Developer/CommandLineTools/usr/share/git-core*. Инструкции по использованию этой функциональной возможности приведены в начале соответствующего файла, предназначенного для используемой ОС.

Теперь вернемся в основную ветвь master и посмотрим, как выглядит рабочий каталог:

```
vagrant@trusty:~/net-auto$ git checkout master
Switched to branch 'master'
vagrant@trusty:~/net-auto$ ls -la
total 40
drwxrwxr-x 3 vagrant vagrant 4096 Jun  7 16:49 .
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 Jun  7 16:49 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 pw.txt
-rwxrwxr-x 1 vagrant vagrant    0 Jun  7 16:34 script.py
-rw-rw-r-- 1 vagrant vagrant  200 Jun  3 20:47 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
vagrant@trusty:~/net-auto$
```

Но здесь нет только что созданного файла *sw6.txt*. Что произошло? Не беспокойтесь, вы ничего не потеряли. Напомним, что выбор (или «извлечение») ветви делает ее активной, следовательно, именно эта ветвь присутствует в рабочем каталоге и доступна для внесения изменений. Файл *sw6.txt* располагается не в основной ветви, а в ветви testing, поэтому после переключения в основную ветвь командой git checkout master этот файл исключается из рабочего каталога. Также напомним, что рабочий каталог – это не то же самое, что репозиторий, таким образом, даже если файл исключен из рабочего каталога, он остается в репозитории. Это легко проверить:

Tlgm: @it_boooks


```
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$ ls -la
total 44
drwxrwxr-x 3 vagrant vagrant 4096 Jun  7 16:53 .
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 Jun  7 16:53 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 pw.txt
-rwxrwxr-x 1 vagrant vagrant    0 Jun  7 16:34 script.py
-rw-rw-r-- 1 vagrant vagrant  200 Jun  3 20:47 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
-rw-rw-r-- 1 vagrant vagrant  221 Jun  7 16:53 sw6.txt
vagrant@jessie:~/net-auto$
```

Приведенные примеры наглядно демонстрируют, как ветви помогают полностью изолировать вносимые изменения от основной ветви. Это самое главное преимущество использования механизма ветвей. Используя одну из ветвей, можно вносить некоторые изменения, тестировать их, а при необходимости удалять (отменять) неудачные варианты, при этом точно знать, что основная ветвь надежно защищена от этих изменений.

Иногда изменения в одной из побочных ветвей становятся настолько важными, что их необходимо сохранить (и каким-то образом перевести в основную ветвь). Например, новый шаблон Jinja продемонстрировал безупречную работу при тестировании в ветви testing, и его следовало бы сделать постоянной частью репозитория. В подобных случаях помогает механизм объединения (merging) ветвей.

Объединение и удаление ветвей

Прежде чем начать изучение работы механизма объединения ветвей, вернемся к содержимому объекта коммит в системе Git. В репозитории из нашего примера внимательно рассмотрим содержимое объекта самого последнего коммита в ветви testing:

```
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$ git cat-file -p b45a2b1
tree f6b5dfbfdbf6bc29f04300c9a82c6936397d9b27
parent 3588c31cbf958fe1a28d5e1f19ace669de99bb8c
author John Smith <john.smith@networktocode.com> 1465317796 +0000
committer John Smith <john.smith@networktocode.com> 1465317796 +0000

Add sw6 configuration
vagrant@jessie:~/net-auto$
```

Попробуем точно определить конкретные элементы полученной информации:

Tlgm: @it_boooks

1. Рассматриваемый здесь коммит указывает на объект дерево с хеш-значением `f6b5df`.
2. Автор внесенного изменения и исполнитель коммита – John Smith.
3. В сообщении коммита указано, что в данный коммит включено добавление файла конфигурации для коммутатора sw6.
4. У рассматриваемого здесь коммита есть родительский коммит с хеш-значением `3588c31`.

Ранее уже отмечалось, что каждый коммит (за исключением самого первого) содержит указатель на родительский коммит. Этот факт наглядно показан на рис. 8.2–8.7, где объекты коммиты указывают на предыдущие коммиты, выполненные в более ранние моменты времени.

При объединении ветвей система Git создает новый объект коммит, обозначаемый термином «коммит объединения» (merge commit), у которого имеются два родителя. Каждый родитель представляет одну из ветвей, участвующих в процессе объединения. При этом Git обеспечивает сохранение и поддержку обратных ссылок на предыдущие коммиты, так что всегда можно «откатиться» к предыдущим версиям.

Внешне (на самом высоком уровне) процесс объединения выглядит следующим образом:

1. Выполняется переключение на ветвь, с которой должна быть объединена другая ветвь. Если необходимо выполнить объединение с основной ветвью, то необходимо сделать активной (переключиться на) ветвь master.
2. Выполняется команда `git merge` с указанием имени ветви, объединяемой с основной ветвью.
3. Формируется сообщение (сообщение для коммита объединения) с описанием изменений, включаемых в объединенную ветвь.

Выполнение «моментальных» объединений

Перейдем к практике: в ветви testing имеется новый файл конфигурации sw6.txt, который отсутствует в основной ветви. Необходимо объединить ветвь testing с основной ветвью.

Сначала нужно убедиться в том, что мы находимся в основной ветви:

```
[vagrant@centos net-auto]$ git branch
  master
* testing
[vagrant@centos net-auto]$ git checkout master
Switched to branch 'master'
[vagrant@centos net-auto]$
```

Теперь выполняется операция объединения, включающая ветвь testing в основную ветвь master:

```
[vagrant@centos net-auto]$ git merge testing
Updating 3588c31..b45a2b1
Fast-forward
 sw6.txt | 12 ++++++++++++
```

Tlgm: @it_boooks


```
 1 file changed, 12 insertions(+)
 create mode 100644 sw6.txt
[vagrant@centos net-auto]$
```

Отметим, что система Git отвечает «моментально». Это признак того, что существовала возможность объединения ветвей с помощью простого воспроизведения набора изменений, внесенных ранее в альтернативную ветвь (testing), как если бы они выполнялись в основной ветви (master). В подобных случаях (простое объединение) вы не увидите дополнительного коммита объединения:

```
vagrant@trusty:~/net-auto$ git log --oneline
b45a2b1 Add sw6 configuration
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ ls -la sw6.txt
-rw-rw-r-- 1 vagrant vagrant 221 Jun  7 20:53 sw6.txt
vagrant@trusty:~/net-auto$
```

Удаление ветви

После объединения ветвей, то есть после включения альтернативной ветви (и всех ее изменений) в основную (или какую-либо другую) ветвь можно удалить эту ветвь с помощью команды `git branch -d` *`имя_ветви`*. Не следует удалять ветвь до объединения, так как можно потерять все изменения, хранящиеся в этой ветви (при попытке удаления необъединенной ветви система Git выведет дополнительный запрос на подтверждение удаления). После объединения изменения надежно сохранены в другой ветви (обычно в основной, но не всегда), следовательно, старую ветвь можно безопасно удалить.

Выполнение объединений с коммитом объединения

Теперь рассмотрим более сложный пример. Сначала создадим новую ветвь для хранения нескольких изменений, касающихся конфигурации коммутатора sw4. Для этого выполняется команда `git checkout -b sw4`. При этом создается новая ветвь и сразу становится активной. После внесения некоторых изменений в файл *sw4.txt* необходимо выполнить команды `git add` и `git commit` для фиксации в индексе и выполнения коммита этих изменений.

Далее выполняется переключение в основную ветвь master (команда `git checkout master`) и вносятся некоторые изменения в другую версию конфигурации коммутатора. Затем изменения фиксируются в индексе и выполняется их коммит в основную ветвь. Что произойдет, если теперь попытаться выполнить объединение ветви sw4 с основной ветвью?

Прежде чем ответить на этот вопрос, посмотрим на объекты коммиты чуть более внимательно. Ниже приведено содержимое объекта самого последнего коммита в ветви sw4:

Tlgm: @it_boooks


```
vagrant@jessie:~/net-auto$ git checkout sw4
Switched to branch 'sw4'
vagrant@jessie:~/net-auto$ git log --oneline HEAD~2..HEAD
40e88b8 Add port descriptions for sw4
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$ git cat-file -p 40e8b8
tree 845b53f6715d73c36d90b3fe3224bfb494853ba5
parent b45a2b162334376f2100974687742de1a23c2594
author John Smith <john.smith@networktocode.com> 1465333657 +0000
committer John Smith <john.smith@networktocode.com> 1465333657 +0000

Add port descriptions for sw4
vagrant@jessie:~/net-auto$
```

Команда `git log --oneline HEAD~2..HEAD` показывает два последних коммита относительно указателя HEAD (который указывает на самый последний коммит в активной ветви). Здесь можно видеть, что этот объект коммита указывает на родительский коммит с хеш-значением `b45a2b1`.

Самый последний коммит в основной ветви:

```
vagrant@jessie:~/net-auto$ git checkout master
Switched to branch 'master'
vagrant@jessie:~/net-auto$ git log --oneline HEAD~2..HEAD
183b8fe Fix sw3 configuration for hypervisor
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$ git cat-file -p 183b8fe
tree 42a59b4058f927ef5af049e581480cd5530bd3b1
parent b45a2b162334376f2100974687742de1a23c2594
author John Smith <john.smith@networktocode.com> 1465333836 +0000
committer John Smith <john.smith@networktocode.com> 1465333836 +0000

Fix sw3 configuration for hypervisor
vagrant@jessie:~/net-auto$
```

Этот коммит, самый последний в основной ветви, также указывает на тот же самый родительский коммит, отмечая место разделения на две ветви.

Теперь выполняется объединение этих ветвей:

```
vagrant@jessie:~/net-auto$ git branch
* master
  sw4
vagrant@jessie:~/net-auto$ git merge sw4
(default Git editor opens to allow user to provide commit message)
Merge made by the 'recursive' strategy.
 sw4.txt | 4 ++++
 1 file changed, 4 insertions(+)
vagrant@jessie:~/net-auto$ git log --oneline HEAD~3..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$
```

Tlgm: @it_boooks

В рассматриваемом случае при объединении ветвей должны быть согласованы изменения в обеих ветвях. Невозможно просто «воспроизвести» изменения из ветви sw4 в основной ветви, потому что основная ветвь содержит собственные изменения. Поэтому система Git создает *коммит объединения* (*merge commit*). Рассмотрим подробнее этот файл:

```
[vagrant@centos net-auto]$ git cat-file -p 81f5963
tree e71cfa26549241b609fa39e69dc51fdafd1d7cb4
parent 183b8fe2d02bbd6b2b7a19ecc39dc9e792fe2e75
parent 40e88b88271b535cceb311bb904d6afac20c15c3
author John Smith <john.smith@networktocode.com> 1465334492 +0000
committer John Smith <john.smith@networktocode.com> 1465334492 +0000

Merge branch 'sw4'
[vagrant@centos net-auto]$
```

Отметим наличие двух родительских коммитов, которые, как можно предположить, представляют коммиты, ранее выполненные в каждой ветви перед объединением sw4 и master. Именно таким способом система Git осуществляет объединение ветвей и согласовывает отношения между коммитами, выполненными за время существования двух различных ветвей.

После включения коммитов из ветви sw4 в основную ветвь альтернативную ветвь можно сразу удалить командой `git branch -d sw4`:

```
[vagrant@centos net-auto]$ git branch -d sw4
Deleted branch sw4 (was 40e88b8).
[vagrant@centos net-auto]$ git branch
* master
[vagrant@centos net-auto]$
```



Удаление необъединенной ветви

Отметим, что удалить необъединенную ветвь все-таки можно с помощью команды `git branch -D` *имя_ветви*. Но при этом будут безвозвратно потеряны все изменения, хранящиеся в удаляемой ветви, поэтому пользуйтесь такой командой внимательно и осторожно.

Подведем итоги:

- для создания ветви используется команда `git branch` *имя_новой_ветви*;
- чтобы сделать ветвь активной, применяется команда `git checkout` *имя_ветви*;
- чтобы одновременно создать ветвь и сделать ее активной, используется команда `git checkout -b` *имя_новой_ветви*;
- для объединения какой-либо ветви с основной (то есть для включения ветви в основную) применяется команда `git merge` *имя_ветви*. При этом основная ветвь master обязательно должна быть активной;
- для удаления ветви после включения ее изменений в другую ветвь используется команда `git branch -d` *имя_ветви*.

Теперь перейдем к изучению использования Git в распределенной среде при одновременной работе группы сотрудников (разработчиков).

Tlgm: @it_boooks

Совместная работа группы сотрудников в системе Git

В разделе «Краткая история создания и развития Git» в начале текущей главы отмечалось, что одной из главных целей при проектировании Git являлось создание полностью распределенной системы. Таким образом, каждому разработчику была необходима возможность работы с полной копией исходного кода, хранящегося в репозитории, а также с полной хронологией репозитория. Если рассматривать распределенную сущность системы Git в совокупности с другими главными целями, установленными при проектировании, – скоростью, простотой, масштабируемостью и полноценной поддержкой нелинейной разработки, реализованной через механизм простых ветвей, то можно понять, почему Git стала самой распространенной и широко используемой системой управления версиями для различных групп пользователей.

Сама по себе система Git может работать как «сервер», предоставляя механизмы обмена данными между различными системами, использующими Git. Git поддерживает разнообразные транспортные протоколы, в том числе Secure Shell (SSH), HTTPS и собственный протокол (на основе протокола TCP, порт 9418). Если Git используется на двух-трех различных системах и необходимо просто поддерживать синхронизацию репозиториев, то для этого не потребуется никакого дополнительного программного обеспечения.

Распределенная сущность системы Git позволила создавать онлайновые сервисы на ее основе. Многие пользователи Git по достоинству оценили преимущества таких сервисов, как GitHub (https://github.com/) и BitBucket (https://bitbucket.org/). Кроме того, существует множество проектов с открытым исходным кодом, в которых совместная работа обеспечивается с помощью Git, например GitLab (https://about.gitlab.com/), Gitblit (http://gitblit.com/index.html) и Djacket (http://djacket.github.io/) (последний проект «по иронии судьбы» размещен на GitHub). Очевидно, что возможности организации совместной работы различных групп пользователей с применением Git и инструментальных средств на основе этой системы практически ничем не ограничены.

В этом разделе будет рассматриваться совместная работа группы сотрудников в системе Git. Методы совместной работы могут быть простыми, как, например, поддержка синхронизации репозиториев на нескольких системах, но, кроме того, будет обсуждаться использование общедоступных онлайн-сервисов на основе Git (основное внимание будет уделено сервису GitHub). В процессе обсуждения вы узнаете о клонировании репозиториев, о методиках работы с Git на удаленных системах, о механизмах передачи (pushing), получения (fetching) и «перетаскивания» (pulling) данных при работе с другими (удаленными) репозиториями, а также об использовании ветвей при совместной групповой работе.

Начнем с изучения простого сценария, в котором используется несколько систем, на которых работает Git, при этом необходимо совместно использовать/синхронизировать один или несколько репозиториев на этих системах.

Tlgm: @it_boooks

Совместная работа в нескольких системах, использующих Git

До настоящего момента в текущей главе создавался набор сетевых конфигураций, скриптов и шаблонов в репозитории Git на одной системе. А что делать, когда необходимо обеспечить доступ к этому репозиторию из другой системы? Пользователь может работать с десктоп-системой в офисе, но дома или в поездках пользоваться ноутбуком. Как организовать работу с репозиторием сетевой автоматизации из обеих систем? Очень просто, если вспомнить о том, что Git представляет собой полноценную распределенную систему.

Насколько это сложнее, чем простое копирование файлов? Попробуем разобраться, что происходит, если просто скопировать репозиторий и его рабочий каталог в другую локацию в той же системе. Сначала выполним команду `git log --oneline HEAD~2..HEAD` в существующем репозитории:

```
vagrant@trusty:~/net-auto$ git log --oneline HEAD~2..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/net-auto$
```

Теперь скопируем репозиторий и его рабочий каталог в другую локацию в той же системе, затем выполним аналогичную команду `git log` и проверим результат:

```
vagrant@trusty:~$ cp -ar net-auto netauto2
vagrant@trusty:~$ cd netauto2
vagrant@trusty:~/netauto2$ git log --oneline HEAD~2..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/netauto2$
```

Содержимое выглядит абсолютно одинаковым. Если продолжить исследование содержимого копии репозитория в каталоге *~/netauto2* с помощью `git ls-tree`, `git cat-file` и других команд, то можно обнаружить, что оба репозитория совершенно одинаковы. Причину этого легко объяснить, если вспомнить, что в системе Git используются хеш-значения SHA1 для идентификации всех элементов содержимого: блобов, деревьев (как объектов) и коммитов (как объектов). Главная характеристика хеш-значений SHA1 – одинаковое содержимое дает одинаковые хеш-значения. Также напомним, что содержимое репозитория Git является неизменяемым (после создания элемента содержимого его нельзя изменить). Совокупность этих характеристик в сочетании с архитектурой Git определеяет возможность копирования репозитория с применением простых инструментов, таких как команда копирования `cp`, и получения в итоге еще одной полной версии репозитория. Именно эта возможность копирования репозиториев с сохранением всех данных и метаданных является самым главным фактором в полностью распределенной сущности системы Git.

Tlgm: @it_boooks

Отметим отсутствие каких-либо связей между копиями, поэтому изменения, внесенные в одну из копий, не отображаются автоматически в другой копии. (При желании это можно проверить, выполнив коммит в любой копии, после чего с помощью команды `git log` нужно проверить содержимое обоих репозиториев.) Для создания связи между репозиториями необходимо воспользоваться специальным механизмом, который называется *remote*.

Установление связи между репозиториями с помощью механизма remote

Механизм remote в системе Git – это просто ссылка на другой репозиторий. Git использует простые указатели весьма активно – выше мы уже рассматривали использование подобных указателей для ветвей и HEAD – на них очень похож указатель типа remote. Таким образом, remote – это простой указатель на другой репозиторий, определяемый по его месту расположения (локации).

Добавим в репозиторий *netauto2* указатель remote, который ссылается на исходный репозиторий в *net-auto*. Для выполнения этой задачи предназначена команда `git remote`:

```
vagrant@jessie:~/netauto2$ git remote
vagrant@jessie:~/netauto2$ git remote add first ~/net-auto
vagrant@jessie:~/netauto2$ git remote
first
vagrant@jessie:~/netauto2$
```

При использовании команды `git remote` без параметров выводится список всех существующих указателей remote. В данном случае указателей пока еще нет. Поэтому следующим нашим шагом будет выполнение команды `git remote add` с двумя параметрами:

- имя используемого удаленного репозитория. Это имя указывается в чисто символьной форме и может быть любым словом (литералом), несущим определенный смысл для пользователя. В нашем примере для удаленного репозитория используется имя `first`;
- место расположения удаленного репозитория. В нашем примере «удаленный» (условно) репозиторий размещен в той же системе, поэтому место расположения указывается просто как путевое имя в файловой системе.

После этого еще раз выполняется команда `git remote`, показывающая, что действительно был добавлен новый удаленный репозиторий.

После определения удаленного репозитория установлена асимметричная связь между двумя удаленными репозиториями. То есть в *netauto2* есть ссылка на *net-auto*, но в обратном направлении ссылки нет. Через такую асимметричную связь можно обмениваться информацией между репозиториями Git. Рассмотрим это взаимодействие более подробно.

Для начала выведем список ветвей, доступных в репозитории *netauto2*. В выполняемую команду добавляется флаг `-a`, смысл которого будет объяснен немного позже:

Tlgm: @it_boooks


```
vagrant@trusty:~/netauto2$ git branch -a
* master
vagrant@trusty:~/netauto2$
```

Следующей выполняется операция *выборки* (*fetch*) (здесь и далее термин выборка (fetch) используется преднамеренно по причине, которая будет обоснована ниже в текущем разделе) информации из удаленного репозитория, сконфигурированного выше. Информация обновляется с помощью команды `git remote update`, затем снова выполняется команда `git branch -a`:

```
vagrant@trusty:~/netauto2$ git remote update first
Fetching first
From /home/vagrant/net-auto
 * [new branch]      master     -> first/master
vagrant@trusty:~/netauto2$ git branch -a
* master
  remotes/first/master
vagrant@trusty:~/netauto2$
```

Отметим, что здесь включена в список новая ветвь. Это особый тип ветви – *удаленная отслеживаемая ветвь* (*remote tracking branch*). В эту ветвь не будут вноситься изменения или коммиты, поскольку это всего лишь ссылка на ветвь, существующую в удаленном репозитории. Обратите внимание на элемент `first` в выведенном имени ветви – это все та же ссылка на символьное имя, присвоенное удаленному репозиторию Git при его добавлении. Здесь необходимо использовать флаг `-a` в команде `git branch`, чтобы вывести удаленные отслеживаемые ветви, которые по умолчанию не выводятся.



Вместо двух последовательно выполняемых команд `git remote add` и `git remote update` можно выполнить выборку информации из удаленного репозитория сразу при его добавлении одной командой `git remote add -f` *имя место_расположения*.

Новая удаленная отслеживаемая ветвь позволяет *передавать* (*transfer*) информацию между репозиториями для поддержки актуальности их состояний. В следующем разделе подробно рассматривается этот процесс.

Получение и объединение информации из удаленных репозиториев

После того как для текущего активного репозитория был сконфигурирован удаленный репозиторий, из которого извлекалась информация, а также были созданы удаленные отслеживаемые ветви, появляется возможность передачи информации между удаленными репозиториями с помощью разнообразных команд `git`. Эти команды можно использовать для синхронизации ветвей или целых репозиториев.

Для наблюдения этого процесса на практике необходимо изменить один из двух репозиториев нашего примера (скажем, *net-auto*) и рассмотреть, как происходит передача информации об изменении в репозиторий *netauto2*.

Сначала в репозитории *net-auto* вносится изменение в файл конфигурации *sw2.txt* и выполняется коммит этого изменения. (Так как все перечисленные

Tlgm: @it_boooks

операции подробно рассматривались выше, здесь они не приводятся. Подсказка: отредактируйте файл, выполните команду `git add`, затем команду `git commit`.)

Проверим наличие нового коммита в репозитории *net-auto* с помощью команды `git log --oneline HEAD~1..HEAD`, затем перейдем в репозиторий *netauto2* и выполним ту же команду `git log`. Списки коммитов различны.

Для получения обновленной информации из репозитория *net-auto* и включения ее в репозиторий *netauto2* можно использовать несколько вариантов действий:

- команда `git remote update` *имя*, которая обновляет *только* заданный репозиторий *имя*;
- команда `git remote update` (без указания имени), которая обновляет *все* репозитории, определенные как удаленные для текущего репозитория;
- команда `git fetch` *имя*, которая обновит информацию (или выполнит выборку информации), взяв ее из заданного удаленного репозитория. В этом отношении команда `git fetch` очень похожа на команду `git remote update`, несмотря на различие в синтаксисе (все особенности и детали применения этих команд подробно описаны на соответствующих страницах руководства man). Отметим, что команда `git fetch` считается «общепринятым» способом извлечения информации из удаленных объектов, в отличие от команды `git remote update`, которую мы использовали выше.

Поэтому мы выбираем выполнение команды `git fetch first`, которая извлекает информацию из репозитория с именем *first*. Вывод результата выполнения этой команды показан ниже (разумеется, хеш-значения SHA различны):

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/vagrant/net-auto
   81f5963..7c2a3e6  master     -> first/master
```

Итак, информация из репозитория *first*/*master* извлечена (из основной ветви репозитория *first*). Но почему команда `git log` в репозитории *netauto2* не показывает полученный результат? Причина в том, что была сделана только *выборка* (*fetch*) (обновление) информации из удаленного репозитория, но эта информация пока еще не стала частью текущего репозитория.



Необходимо предупредить читателей о некорректности использования термина «pull» («перетаскивание») для обозначения простого извлечения (retrieving) информации из удаленного репозитория. В системе Git концепция «перетаскивания» (pulling) из удаленного репозитория имеет чрезвычайно особый смысл и реализована с помощью отдельной команды (концепция и команда будут рассматриваться немного позже). Рекомендуется выработать привычку использования термина «извлечение» (fetching или retrieving) для обозначения операции получения информации из удаленного репозитория.

Если изменения из другого репозитория только извлечены (fetched), но пока еще не стали частью текущего репозитория, то необходим способ включения

Tlgm: @it_boooks

извлеченных данных в текущий репозиторий. Изменения в удаленном репозитории хранятся в отдельной ветви, то есть отдельно от определенной по умолчанию главной ветви в текущем репозитории. Но как передать изменения в одной ветви в другую ветвь? Ответ очевиден – необходимо объединить (merge) эти ветви:

```
vagrant@jessie:~/netauto2$ git checkout master
Already on 'master'
vagrant@jessie:~/netauto2$ git merge first/master
Updating 81f5963..7c2a3e6
Fast-forward
 sw2.txt | 7 +++++++
 1 file changed, 7 insertions(+)
vagrant@jessie:~/netauto2$
```

Здесь можно видеть, что изменения, внесенные в ветвь *first/master* (в основную ветвь удаленного репозитория *first*), объединены (по ускоренной методике) с основной ветвью (master) текущего репозитория. Так как применялась ускоренная методика (fast-forward), коммит объединения не фиксируется. Теперь оба репозитория синхронизированы.



Если вы заметили, что применение команд `git fetch` и `git merge` в основной ветви обоих репозиториев не всегда гарантирует полную синхронизацию этих репозиториев, то ваша наблюдательность достойна уважения – вы совершенно правы. В действительности синхронизированы только основные ветви двух репозиториев. Чтобы обеспечить полную синхронизацию репозиториев, необходимо выполнить эту операцию для всех существующих ветвей.

«Перетаскивание» информации из удаленных репозиториев

Зачем нужен двухэтапный процесс с выполнением сначала команды `git fetch`, затем команды `git merge`? Главная причина: может потребоваться обзор изменений, принимаемых из удаленного репозитория, перед операцией объединения (то есть включения этих изменений в текущий репозиторий). В некоторых случаях может оказаться, что вы не готовы применить сторонние изменения в текущем репозитории.

Но для выполнения такого двухэтапного процесса существует более быстрый альтернативный способ, как и для многих процессов в системе Git. Если нужно извлечь изменения и внести их в текущий репозиторий в одной операции, то можно воспользоваться командой `git pull` *имя*, где *имя* соответствует имени удаленного репозитория, из которого необходимо получить изменения и включить их в текущую ветвь. Команда `git pull` просто объединяет операции `git fetch` и `git merge`.

В предыдущем разделе рассматривалось получение изменений из репозитория *net-auto* в репозиторий *netauto2*, но можно ли выполнить обратную операцию? Ранее отмечалось, что при добавлении удаленного репозитория в *netauto2* создается асимметричное отношение, то есть теперь *netauto2* знает о существовании *net-auto*, но обратное неверно. В подобной ситуации – един-

Tlgm: @it_boooks

ственный пользователь желает поддерживать синхронизацию между двумя репозиториями, размещенными на различных системах, – оптимальным решением могло бы стать добавление *netauto2* как удаленного репозитория для *net-auto* и последующее выполнение команд `git fetch` и `git merge` для передачи изменений в обоих направлениях. Графическая схема такого взаимодействия приведена на рис. 8.9.

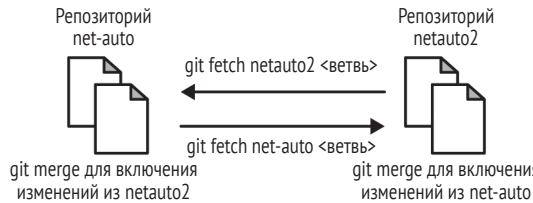


Рис. 8.9 ❖ Использование команд git fetch и git merge для обмена данными между репозиториями

Система Git почти всегда предлагает несколько способов выполнения тех или иных операций, и эти способы могут оказаться как полезными (именно в этом и заключается чрезвычайная гибкость системы), так и разочаровывающими (поэтому система не ограничивается единственным способом выполнения какой-либо операции). Двунаправленная передача информации между репозиториями Git, описанная в текущем разделе, представляет собой одну из таких областей, в которых существует более одного способа выполнения задачи. Для пользователей, только начинающих освоение системы Git, описанный выше способ является, вероятно, самым простым для понимания.

Текущий раздел начинался с вопроса: «Как я могу воспользоваться своим репозиторием для автоматизации сети из нескольких различных систем?» В разделе были показаны способы создания копий репозитория, применение механизма remote для установления связи с удаленными репозиториями, а также использование разнообразных команд Git для передачи данных между репозиториями. В следующих двух разделах рассматривается более простой способ копирования и связывания репозиториев, и мы соответствующим образом распределим нашу рабочую модель между несколькими системами.

Клонирование репозиториев

В предыдущих разделах рассматривалась возможность создания простой копии репозитория в другой локации с последующим использованием команды `git remote` для установления удаленного репозитория, что позволяло передавать информацию между репозиториями.

Выполнение этого процесса не вызывает каких-либо затруднений, но, может быть, существует еще более простой способ? Оказывается, существует – клонирование (cloning) репозитория с помощью команды `git clone`. Рассмотрим этот способ подробнее.

Общий синтаксис команды: `git clone` *`репозиторий каталог`*.

Tlgm: @it_boooks

Здесь *репозиторий* – это место расположения (локация) клонируемого репозитория, а *каталог* – имя каталога (его указывать не обязательно), в котором предполагается разместить клон исходного репозитория. Если параметр *каталог* отсутствует, то Git разместит клон репозитория в каталоге с именем, совпадающим с именем исходного репозитория. Добавление параметра *каталог* в команду `git clone` предоставляет пользователю право выбора места размещения создаваемого клона репозитория.

Чтобы продемонстрировать работу команды `git clone` на практике, сначала удалим репозиторий *netauto2*. В нем не должно быть каких-либо новых изменений, но если они были внесены, то вы уже знаете, как передать эти изменения в исходный репозиторий *net-auto*. (Подсказка: добавить удаленный репозиторий, извлечь (fetch) новые изменения и включить их (merge) в исходный репозиторий.)

```
vagrant@trusty:~$ rm -rf netauto2
vagrant@trusty:~$ git clone ~/net-auto na-clone
Cloning into 'na-clone'...
done.
vagrant@trusty:~$ cd na-clone
vagrant@trusty:~/na-clone$ git log --oneline HEAD~2..HEAD
7c2a3e6 Update sw2 configuration
81f5963 Merge branch 'sw4'
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/na-clone$
```

Здесь можно видеть, что команда `git clone` создает копию репозитория точно так же, как это делалось вручную в предыдущем разделе. Но команда не ограничивается только созданием копии – выполним команду `git remote` в новом репозитории-клоне:

```
vagrant@trusty:~/na-clone$ git remote -v
origin
vagrant@trusty:~/na-clone$
```

В этом и состоит преимущество использования команды `git clone`, по сравнению с выполнением операций вручную, рассматриваемым выше, – команда автоматически создает в репозитории-клоне обратный указатель remote на исходный репозиторий как на удаленный. Более того, при этом автоматически создается удаленная отслеживаемая ветвь (это можно проверить командой `git branch -a` или `git branch -r`). Благодаря этим дополнительным операциям команду `git clone` можно считать наиболее предпочтительным механизмом для создания полной копии, связанной с исходным репозиторием (то есть клона).

Прежде чем продолжить, уделим немного внимания ветви origin, которая была автоматически создана при выполнении команды `git clone`. Напомним, что имя удаленного репозитория чисто символическое (и может выбираться произвольно), но имя origin имеет особое значение для Git. Его можно считать именем удаленного объекта, назначаемым по умолчанию. В том случае, когда вы работаете с несколькими удаленными объектами (разумеется, такая воз-

Tlgm: @it_boooks

можность определенно существует) и выполняете команду `git fetch` без указания конкретного удаленного объекта, Git по умолчанию выбирает origin.

В качестве примера использования нескольких удаленных объектов можно привести процесс написания данной книги: одним из удаленных объектов был репозиторий сервиса GitHub, кроме того, существовал репозиторий авторов в издательстве O'Reilly. Ниже приведен вывод команды `git remote -v`, выполненной из репозитория одного из авторов:

```
oreilly     git@git.atlas.oreilly.com:oreillymedia/network-automation.git (fetch)
oreilly     git@git.atlas.oreilly.com:oreillymedia/network-automation.git (push)
origin      https://github.com/jedelman8/network-automation-book.git (fetch)
origin      https://github.com/jedelman8/network-automation-book.git (fetch)
```

Теперь мы полностью готовы к завершающему этапу, на котором все полученные до настоящего момента знания будут практически применены для распространения нашей рабочей модели, рассматриваемой на примерах, по репозиториям на различных системах.

Расширение рабочей модели из примеров для использования на нескольких системах

При обсуждении концепции создания удаленного объекта Git (см. раздел «Установление связи между репозиториями с помощью механизма remote») было отмечено, что каждый удаленный объект имеет два атрибута: имя (произвольное и условное) и место расположения (локация). До сих пор рассматривались удаленные объекты, размещенные в одной системе, но Git благодаря своей распределенной сущности поддерживает работу с реально удаленными объектами в различных системах с использованием разнообразных протоколов.

Например, удаленный объект, размещенный в той же системе, что и исходный, использует следующие типы локаций:

```
/path/to/git/repository
file:///path/to/git/repository
```

Но реально удаленный объект также может использовать разнообразные протоколы для получения доступа к репозиторию на другой системе:

```
git://host.domain.com/path/to/git/repository
ssh://[user@]host.domain.com/path/to/git/repository
http://host.domain.com/path/to/git/repository
https://host.domain.com/path/to/git/repository
```

Префикс `git://` обозначает собственный протокол системы Git, в котором не предусмотрена аутентификация, следовательно, обеспечивается анонимный доступ (вообще говоря, это доступ только для чтения или с защитой от записи). Префикс `ssh://` обозначает протокол Secure Shell (SSH); в действительности система Git применяет протокол с туннелингом через SSH для обеспечения доступа с выполнением процедуры аутентификации. Два последних варианта – доступ по протоколам HTTP и HTTPS соответственно.

Tlgm: @it_boooks

Это означает, что рабочую модель, которая рассматривалась в примерах текущего раздела можно без затруднений расширить для использования на нескольких системах с применением наиболее подходящего сетевого протокола. В этом разделе главное внимание будет уделено протоколу SSH, а в конце главы будут приведены примеры использования протокола HTTPS для работы с общедоступными сервисами Git.

Вернемся к нашему примеру и предположим, что необходимо обеспечить работу с ранее созданным репозиторием автоматизации сети десктоп-системы и ноутбука в динамическом режиме. Допустим, что обе системы поддерживают протокол SSH (то есть обе системы работают под управлением Linux, macOS или какого-либо другого варианта Unix). Можно порекомендовать сначала создать конфигурацию для беспарольной аутентификации по протоколу SSH, в основном с применением SSH-ключей. Это позволит выполнять различные команды `git` без предоставления пароля для каждой команды. Создание конфигурации SSH не относится к теме данной книги, но этот процесс очень хорошо документирован, и документацию легко найти в интернете.

После этого создаются необходимые удаленные объекты Git. Исходный репозиторий уже существует на десктоп-системе (для этого примера в качестве ОС для десктоп-системы используется Ubuntu версии Trusty Tahr). На нотбуке репозитория пока нет (ноутбук работает под управлением ОС Debian Jessie). Поэтому для начала необходимо создать клон репозитория на нутбуке:

```
vagrant@jessie:~$ git clone ssh://trusty.domain.com/~/net-auto net-auto
Cloning into 'net-auto'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 32 (delta 12), reused 0 (delta 0)
Receiving objects: 100% (32/32), 2.99 KiB | 0 bytes/s, done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
vagrant@jessie:~$
```

Репозиторий с десктопа (*trusty.domain.com*) копируется на ноутбук (с размещением в заданном каталоге; в нашем примере это каталог *net-auto*), создается удаленный объект Git с именем origin, указывающий на исходный клонируемый объект, также создаются удаленные отслеживаемые ветви. Это можно проверить, выполнив команду `git remote` для просмотра списка удаленных объектов и команду `git branch -r` для просмотра списка удаленных отслеживаемых ветвей.

Если более предпочтительным является имя удаленного объекта, более точно определяющее его место расположения, то можно заменить присваемое по умолчанию имя origin на более информативное имя. Выполним переименование удаленного объекта так, чтобы новое имя отображало происхождение от источника, размещенного на десктоп-системе Ubuntu:

```
vagrant@jessie:~/net-auto$ git remote
origin
vagrant@jessie:~/net-auto$ git remote rename origin trusty
```

Tlgm: @it_boooks


```
vagrant@jessie:~/net-auto$ git remote
trusty
vagrant@jessie:~/net-auto$
```

Вернемся в систему Ubuntu, где необходимо создать удаленный объект, соответствующий репозиторию на ноутбуке под управлением Debian. Удаленный репозиторий уже существует, поэтому использовать команду `git clone` нельзя. Вместо этого нужно вручную добавить удаленный объект, затем получить от него информацию, чтобы создать удаленные отслеживаемые ветви:

```
vagrant@trusty:~/net-auto$ git remote add jessie ssh://jessie.domain.com/~/net-auto
vagrant@trusty:~/net-auto$ git remote
jessie
vagrant@trusty:~$ git fetch jessie
From ssh://jessie.domain.com/~/net-auto
 * [new branch]      master     -> jessie/master
vagrant@trusty:~/net-auto$ git branch -r
  jessie/master
vagrant@trusty:~/net-auto$
```

Теперь репозитории созданы в обеих системах, указатели remote в каждой системе ссылаются на удаленные объекты в другой системе, также созданы необходимые удаленные отслеживаемые ветви с каждой стороны. После этого рабочий процесс (поток) выполняется в точности так же, как это было описано в предыдущих разделах:

1. Вносятся изменения на одной из систем (но не на обеих одновременно), выполняется коммит этих изменений в репозиторий. В идеальном случае рекомендуется работать *только* в ветвях, отличающихся от основной ветви master.
2. После перехода в другую систему выполняются команды `git fetch` и `git merge` для получения внесенных изменений и внесения изменений из удаленных ветвей в локальные. (Если нет необходимости проверять изменения перед операцией объединения, то можно воспользоваться командой `git pull`.) Операцию объединения изменений из ветвей разных систем необходимо выполнить перед началом работы на локальной системе, это обязательное условие.
3. При необходимости повторять выполнение пунктов 1 и 2 для обеспечения синхронного состояния обеих систем и актуальности данных в репозиториях.

Описанная выше методика вполне подходит для одного разработчика, в распоряжении которого находятся две различные системы, но что делать, если разработчиков несколько? Конечно, можно сформировать «полносвязную сеть» из удаленных объектов Git и удаленных отслеживаемых ветвей, но такая структура быстро становится чрезвычайно громоздкой и весьма неудобной в управлении. В подобных случаях проблема решается с помощью совместно используемого репозитория, как будет продемонстрировано в следующем разделе.

Tlgm: @it_boooks

Совместно используемый репозиторий

Осваивая работу с удаленными объектами Git с использованием команды `git remote`, вы, вероятно, обратили внимание на флаг `-v`, позволяющий выводить более подробную информацию. Например, при выполнении команды `git remote -v` на одной из систем, сконфигурированных в предыдущем разделе, выводится следующая информация:

```
trusty  ssh://trusty.domain.com/~/net-auto (fetch)
trusty  ssh://trusty.domain.com/~/net-auto (push)
```

Выводится полное описание места расположения удаленного репозитория, что очень удобно и полезно. Операция получения информации из удаленного репозитория с помощью команды `git fetch` рассматривалась выше, но что можно сказать о концепции ввода push («проталкивание»)?

До сих пор обсуждалась только концепция извлечения информации из удаленного репозитория в локальный с использованием таких команд, как `git remote update`, `git fetch` и `git pull`. Тем не менее существует возможность передачи или отправки изменений (в системе Git используется более «жесткий» термин push (проталкивание)) в удаленный репозиторий из локального. Но в таких случаях настоятельно рекомендуется создать ситуацию, в которой удаленный репозиторий должен быть «неполным» репозиторием.

Что такое «неполный» репозиторий (bare repository)? Проще говоря, неполный репозиторий – это репозиторий Git без рабочего каталога. (Напомним, что термин «рабочий каталог» в системе Git имеет весьма особенный смысл, при этом не имеет ничего общего с текущим каталогом.) До сих пор при рассмотрении репозиториев Git и во всех примерах подразумевалось непременное наличие рабочего каталога, поскольку любой пользователь намерен активно работать с репозиторием. Пользователю должен быть предоставлен некоторый способ взаимодействия с содержимым репозитория, и рабочий каталог является именно таким способом взаимодействия.

Причина, по которой не рекомендуется передача изменений в обычный, то есть «полный», репозиторий (репозиторий с рабочим каталогом): операция передачи (push) не обновляет рабочий каталог. Вернемся к предыдущему примеру, в котором две системы были сконфигурированы как удаленные друг для друга с созданием удаленных отслеживаемых ветвей в обеих системах, и рассмотрим потенциальные проблемы при выполнении операции push.

1. Предположим, что вы корректный пользователь Git, работающий с одной из ветвей. Назовем эту ветвь *new-feature*. Новая функциональная возможность зарегистрирована в индексе первой системы, поэтому содержимое ветви new-feature находится в рабочем каталоге. В конце рабочего дня в офисе изменения еще не завершены и остаются в рабочем каталоге, но вы выполнили коммит нескольких других изменений.
2. Вы получаете изменения из второй системы, проверяете их, затем объединяете эти полученные изменения с локальной ветвью new-feature

Tlgm: @it_boooks

и продолжаете работу. Вам известно, что изменения, для которых коммит пока еще не выполнен, невозможно увидеть в рабочем каталоге первой системы, но это не представляет особой проблемы. Пока все идет хорошо.

3. Дома к вечеру вы наконец завершаете некоторый этап работы и принимаете решение о передаче (push) сделанных изменений в ветвь new-feature основной рабочей системы.
4. На следующий день вы приходите в офис и приступаете к работе. Изменения, для которых не был выполнен коммит, остались в рабочем каталоге, но вы не видите изменений, которые были переданы (pushed) с домашней системы накануне вечером. Что же произошло?

В этом и заключается проблема передачи (pushing) изменений в «полный» (non-bare) репозиторий: изменения передаются («проталкиваются») сразу в удаленный репозиторий, но рабочий каталог при этом не обновляется. Поэтому вы не увидели изменений, переданных накануне. Чтобы получить возможность увидеть эти изменения, необходимо выполнить команду `git reset --hard HEAD`, которая *отбросит* все изменения в рабочем каталоге, чтобы показать переданные (pushed) изменения. Ситуация не из лучших.

Использование неполного репозитория (bare repository) устраняет описанную выше проблему, но вместе с тем лишает пользователя возможности интерактивно работать с таким репозиторием. Но это, вероятно, вполне подходящее решение для репозитория, который совместно использует несколько разработчиков.

Чтобы создать новый неполный репозиторий, нужно просто добавить флаг `--bare` в команду `git init`:

```
vagrant@trusty:~$ git init --bare shared-repo.git
Initialized empty Git repository in /home/vagrant/shared-repo.git/
vagrant@trusty:~$ git init non-bare-repo
Initialized empty Git repository in /home/vagrant/non-bare-repo/.git/
```

Отметим различие в сообщениях, выводимых командой `git init` с флагом `--bare` и без него. При создании обычного (non-bare) репозитория реальный репозиторий Git размещается в подкаталоге *.git*, и в сообщении это отображено. Но при создании неполного (bare) репозитория нет рабочего каталога, поэтому реальный репозиторий Git размещается прямо в заданном каталоге.



По общепринятому соглашению имя неполного (bare) репозитория оканчивается суффиксом .git, хотя это не является обязательным требованием.

Но в нашем примере уже имеется существующий репозиторий, поэтому необходимо каким-то способом преобразовать его в неполный репозиторий, чтобы обеспечить совместную работу с ним для нескольких пользователей. В системе Git предусмотрена и такая ситуация: можно воспользоваться командой `git clone`, чтобы для существующего репозитория создать клон в виде нового неполного репозитория.

Tlgm: @it_boooks


```
vagrant@trusty:~$ git clone --bare net-auto na-shared.git
Cloning into bare repository 'na-shared.git'...
done.
vagrant@trusty:~$
```

При выполнении команды `git clone --bare` система Git не добавляет новых удаленных объектов и удаленных отслеживаемых ветвей. Такое решение не лишено смысла, поскольку удаленные объекты и удаленные отслеживаемые ветви в основном необходимы лишь для прямого интерактивного взаимодействия с репозиторием. Для неполного репозитория прямое интерактивное взаимодействие не предусматривается. Вы будете использовать клон на другой системе, в котором будут присутствовать удаленные объекты и удаленные отслеживаемые ветви.

Вернемся к нашему примеру с двумя системами (с репозиториями в системах Trusty и Jessie и указателями remote, ссылающимися на другую систему) и выполним преобразование с созданием совместно используемого неполного репозитория на третьей системе. Третьей системой становится CentOS, предназначенная для обеспечения работы совместно используемого репозитория.

Сначала необходимо создать неполный репозиторий в системе CentOS. Здесь находит применение команда `git clone --bare`:

```
[vagrant@centos ~]$ git clone --bare ssh://trusty.domain.com/~/net-auto na-shared.git
Cloning into bare repository 'na-shared.git'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (30/30), done.
Receiving objects: 100% (32/32), done.
remote: Total 32 (delta 12), reused 0 (delta 0)
Resolving deltas: 100% (12/12), done.
[vagrant@centos ~]$
```

После этого можно клонировать этот неполный репозиторий в две рабочие системы. Сначала в систему Ubuntu Trusty:

```
vagrant@trusty:~$ git clone ssh://centos.domain.com/~vagrant/na-shared.git na-shared
Cloning into 'na-shared'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 32 (delta 12), reused 32 (delta 12)
Receiving objects: 100% (32/32), done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
vagrant@trusty:~$ cd na-shared
vagrant@trusty:~/na-shared$ git remote -v
origin  ssh://centos.domain.com/~vagrant/na-shared.git (fetch)
origin  ssh://centos.domain.com/~vagrant/na-shared.git (push)
vagrant@trusty:~/na-shared$ git branch -r
  origin/HEAD -> origin/master
  origin/master
vagrant@trusty:~/na-shared$ git log --oneline HEAD~2..HEAD
7c2a3e6 Update sw2 configuration
81f5963 Merge branch 'sw4'
```

Tlgm: @it_boooks


```
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/na-shared$
```

Здесь можно видеть, что команда `git clone` выполняет клонирование с созданием неполного репозитория, затем происходит возврат в систему Ubuntu для сохранения всех данных и метаданных в этом репозитории. При этом система Git автоматически создает удаленные объекты и удаленные отслеживаемые ветви. (При необходимости можно проверить хронологию Git, выполнив команду `git log` в новом репозитории `na-shared` и в старом репозитории `net-auto`, продолжающем существовать в системе Ubuntu.)

Далее аналогичные действия выполняются в системе Debian Jessie:

```
vagrant@jessie:~$ git clone ssh://centos.domain.com/~vagrant/na-shared.git
na-shared
Cloning into 'na-shared'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 32 (delta 12), reused 32 (delta 12)
Receiving objects: 100% (32/32), done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
vagrant@jessie:~$ cd na-shared
vagrant@jessie:~/na-shared$ git remote -v
origin  ssh://centos.domain.com/~vagrant/na-shared.git (fetch)
origin  ssh://centos.domain.com/~vagrant/na-shared.git (push)
vagrant@jessie:~/na-shared$ git branch -r
  origin/HEAD -> origin/master
  origin/master
vagrant@jessie:~/na-shared$
```

Теперь на обеих рабочих системах существует новый репозиторий *na-shared*, поэтому можно спокойно удалить старый репозиторий *net-auto* командой `rm -rf net-auto`.

После всех проделанных операций рабочий процесс (поток) выглядит следующим образом:

1. Сохраняется необходимость работы с почти исключительным доступом в ветвях, отличающихся от основной ветви master. Это становится особенно важным при работе нескольких пользователей в одном совместно используемом репозитрии.
2. Перед началом работы в локальном клоне на любой системе необходимо выполнить команду `git fetch` для получения всех изменений, существующих в совместно используемом репозитории, но не в локальном клоне. При необходимости полученные изменения объединяются с локальными ветвями с помощью команды `git merge`.
3. Вносятся изменения в локальный репозиторий, и выполняется коммит этих изменений в локальный клон.
4. Изменения «проталкиваются» в совместно используемый репозиторий с помощью команды `git push`.

Tlgm: @it_boooks

Дополнительные разъяснения по команде `git push` не требуются, поскольку эта команда подробно рассматривалась ранее.

Ввод изменений в совместно используемый репозиторий

Теперь у нас есть неполный репозиторий, и можно вводить («проталкивать» – push) изменения в удаленный объект, используя для этого команду `git push`. Общий синтаксис: `git push` *`remote branch`*, здесь *`remote`* – имя удаленного объекта Git, а *`branch`* – имя ветви, в которую должны «проталкиваться» изменения.

Чтобы продемонстрировать этот процесс на практике, внесем некоторые изменения в репозиторий автоматизации сети, размещенный в системе Debian Jessie. Добавляется шаблон Jinja с именем *hv-tor-config.j2*, определяющий основную конфигурацию для основного коммутатора в стойке, к которому подключаются гипервизоры.

Поскольку нет необходимости работать в основной ветви, сначала создается новая ветвь для хранения вносимых изменений:

```
vagrant@jessie:~/na-shared$ git checkout -b add-sw-tmpl
Switched to a new branch 'add-sw-tmpl'
vagrant@jessie:~/na-shared$
```

После добавления файла в рабочий каталог (файл создается вручную с нуля или копируется с какого-либо существующего источника) изменения фиксируются в индексе, и выполняется их коммит:

```
vagrant@jessie:~/na-shared$ git add hv-tor-config.j2
vagrant@jessie:~/na-shared$ git commit -m "Add Jinja template for TOR config"
[add-sw-tmpl 8cbbe6f] Add Jinja template for TOR config
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@jessie:~/na-shared$
```

После этого изменения вводятся («проталкиваются» – push) в удаленный объект origin, который указывает на ранее созданный совместно используемый (неполный) репозиторий:

```
vagrant@jessie:~/na-shared$ git push origin add-sw-tmpl
Counting objects: 3, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 423 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ssh://centos.domain.com/~vagrant/na-shared.git
 * [new branch]      add-sw-tmpl -> add-sw-tmpl
vagrant@jessie:~/na-shared$
```

Это позволяет сотрудникам одной группы и прочим участникам проекта, привлекаемым к совместной работе, в дальнейшем извлекать внесенные изменения и размещать их на своих системах. Нужно всего лишь воспользоваться командой `git fetch` для получения изменений, создать локальную ветвь, соответствующую удаленной отслеживаемой ветви, затем проверить полученные

Tlgm: @it_boooks

изменения любым приемлемым методом. Ниже демонстрируется использование команды `git diff`, хотя это и не самый удобный способ с учетом того, что единственным внесенным изменением было добавление одного нового файла.

```
vagrant@trusty:~/na-shared$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://centos.domain.com/~vagrant/na-shared
 * [new branch]      add-sw-tmpl -> origin/add-sw-tmpl
vagrant@trusty:~/na-shared$ git checkout --track -b add-sw-tmpl origin/add-sw-tmpl
Branch add-sw-tmpl set up to track remote branch add-sw-tmpl from origin.
Switched to a new branch 'add-sw-tmpl'
vagrant@trusty:~/na-shared$ git diff master..HEAD
diff --git a/hv-tor-config.j2 b/hv-tor-config.j2
new file mode 100644
index 0000000..989d723
--- /dev/null
+++ b/hv-tor-config.j2
@@ -0,0 +1,15 @@
+interface ethernet0
+  switchport mode access vlan {{ mgmt_vlan_id }}
+
+interface ethernet1
+  description Connected to {{ server_uplink_1 }}
+  switchport mode trunk
+
+interface ethernet2
+  description Connected to {{ server_uplink_2 }}
+  switchport mode trunk
+
+interface ethernet3
+  description Connected to {{ server_uplink_3 }}
+  switchport mode trunk
+
vagrant@trusty:~/na-shared$
```

После согласования корректности внесенных изменений со всеми членами рабочей группы можно объединить эти изменения с основной ветвью. Сначала выполняется локальное объединение:

```
vagrant@jessie:~/na-shared$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'
vagrant@jessie:~$ git merge add-sw-tmpl
Updating 7c2a3e6..8cbbe6f
Fast-forward
 hv-tor-config.j2 | 15 +++++++++++++++
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@jessie:~/na-shared$
```

Tlgm: @it_boooks

Это моментальное или оперативное (fast-forward) объединение, поэтому объединяющий коммит отсутствует. Теперь необходимо внести изменения в совместно используемый репозиторий:

```
vagrant@jessie:~/na-shared$ git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To ssh://centos.domain.com/~vagrant/na-shared.git
   7c2a3e6..8cbbe6f  master -> master
vagrant@jessie:~/na-shared$
```

Далее локальная ветвь изменений удаляется (ее часто называют также *ветвью обновлений* (*feature branch*) или *обсуждаемой ветвью* (*topic branch*)), и это изменение также вносится в совместно используемый репозиторий:

```
vagrant@jessie:~/na-shared$ git branch -d add-sw-tmpl
Deleted branch add-sw-tmpl (was 8cbbe6f).
vagrant@jessie:~/na-shared$ git push origin --delete add-sw-tmpl
To ssh://centos.domain.com/~vagrant/na-shared.git
 - [deleted]         add-sw-tmpl
vagrant@jessie:~/na-shared$
```

После этого все члены рабочей группы могут получить изменения, внесенные в основную ветвь, удалить ранее созданные локальные ветви, затем удалить удаленную отслеживаемую ветвь, которая больше не нужна. Для этого используется команда `git fetch --prune`:

```
vagrant@trusty:~/na-shared$ git pull origin master
From ssh://centos.domain.com/~vagrant/na-shared
 * branch            master     -> FETCH_HEAD
   7c2a3e6..8cbbe6f  master     -> origin/master
Updating 7c2a3e6..8cbbe6f
Fast-forward
hv-tor-config.j2 | 15 +++++++++++++++
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@trusty:~/na-shared$ git fetch --prune origin
From ssh://192.168.100.12/~vagrant/na-shared
 x [deleted]         (none)     -> origin/add-sw-tmpl
vagrant@trusty:~/na-shared$ git branch -d add-sw-tmpl
Deleted branch add-sw-tmpl (was 8cbbe6f).
vagrant@trusty:~/na-shared$
```

Здесь применяется новая команда `git fetch --prune`, которая используется для ликвидации удаленной отслеживаемой ветви, когда эта ветвь перестает существовать в удаленном объекте remote. В приведенном примере убрана удаленная отслеживаемая ветвь для *origin/add-sw-tmpl*, как указано в информации, выводимой этой командой.



При изучении Git требуется немало терпения

Для тех, кто только начинает изучение системы Git, все описанное выше может показаться весьма сложным. Но это обычное явление – все начинающие пользователи Git испытывают похожие чувства в той или иной степени (возможно, Линус является исклю-

Tlgm: @it_boooks

чением). Не торопитесь, запаситесь терпением. При активном использовании Git очень скоро команды становятся более привычными и не такими уж сложными. Но на начальном этапе освоения весьма полезно иметь под рукой краткий справочник по всем командам с описанием их синтаксиса.

Прежде чем перейти к последней теме – совместная работа с использованием онлайновых сервисов на основе Git, – подведем краткие итоги по текущему разделу:

- в системе Git используется концепция удаленных объектов (remote) для создания связей между репозиториями. Для работы с удаленными объектами предназначена команда `git remote`. Удаленный объект может указывать на некоторую локацию в файловой системе или на какой-либо узел сети, например на другую систему, доступную по протоколу SSH;
- для извлечения изменений из удаленного репозитория и размещения их в своем локальном репозитории используется команда `git fetch` *`remote`*;
- при работе с удаленными репозиториями в системе Git основным средством являются ветви. Специализированные ветви, называемые удаленными отслеживаемыми ветвями, автоматически создаются при выполнении команды `git fetch` для получения изменений;
- изменения, полученные из удаленного репозитория, могут быть внесены (объединены – merged) в локальный репозиторий точно так же, как любая другая ветвь с помощью команды `git merge`;
- если последовательное выполнение двух команд `git fetch` и `git merge` неудобно, то можно воспользоваться командой `git pull`;
- команда `git push` используется для ввода (проталкивания – push) изменений в удаленный репозиторий, но при этом удаленный репозиторий должен быть неполным (bare) репозиторием;
- применение неполного репозитория в качестве центрального совместно используемого репозитория позволяет нескольким пользователям одновременно работать с одним репозиторием. Изменения вносятся через специализированные ветви с помощью команд `git push`, `git fetch`, `git merge` и `git pull`.

Последний раздел текущей главы основан на материале, изученном в предыдущих разделах, и главное внимание в нем уделено использованию онлайновых сервисов на основе Git для организации совместной работы группы пользователей.

Совместная работа с использованием онлайновых сервисов на основе Git

Совместная работа группы пользователей, использующих онлайновые сервисы на основе Git, в основном почти не отличается от рабочего процесса, описанного в предыдущем разделе. Применяются те же концепции – клонирование для создания копий репозиториев, использование удаленных объектов (remotes) и удаленных отслеживаемых ветвей, работа в специальных ветвях

Tlgm: @it_boooks

для организации обмена изменениями с другими пользователями в одном репозитории. Даже команды те же самые: `git fetch`, `git push`, `git merge` и `git pull`.

Тем не менее следует отметить некоторые различия, заслуживающие внимания. Для краткости мы сосредоточим внимание на использовании GitHub в качестве онлайнового сервиса на основе Git. В текущем разделе рассматриваются две основные темы:

- ответвление (forking) репозиториев;
- запросы на включение изменений (pull requests).

Начнем с ответвлений репозиториев.

Ответвление репозиториев

Ответвление (*forking*) в проекте GitHub – это, по существу, то же самое, что клонирование репозитория Git. (В этом разделе термины «проект» и «репозиторий» используются как взаимозаменяемые, поскольку в данном контексте обозначают одно и то же.) При ответвлении репозитория GitHub выполняется команда, адресованная серверам GitHub, для клонирования указанного репозитория в вашу пользовательскую учетную запись. В момент создания ответвление представляет собой абсолютно полную копию исходного репозитория, включая все содержимое и хронологию коммитов. После завершения ответвления исходного репозитория в вашу учетную запись состояние похоже на то, как если бы была выполнена команда `git clone` из командной строки – сохраняются связи с исходным проектом, во многом похожие на удаленный объект (remote) Git. (Но эти удаленные объекты остаются невидимыми для пользователя.) Главное отличие состоит в том, что ответвление репозитория на GitHub не создает локальную копию исходного репозитория, поэтому сохраняется необходимость выполнения команды `git clone` для клонирования ответвленной копии на вашей локальной системе. Это процедура будет рассматриваться ниже.

Так зачем же выполняется ответвление репозитория? На крупных онлайновых сервисах, таких как GitHub, хранятся и поддерживаются сотни тысяч репозиториев. Каждый из этих репозиториев связан с идентификатором пользователя (user ID) GitHub. Идентификатор пользователя позволяет точно определить, кто имеет право вносить изменения в конкретный репозиторий и у кого такого права нет. Допустим, вы обнаружили репозиторий, в который хотели бы внести свой вклад. Возможно, владелец репозитория вас не знает (вполне вероятная ситуация) и не доверяет вам в степени, достаточной для того, чтобы включить вас в группу, работающую с этим репозиторием. Но если у вас имеется собственная копия такого репозитория, вы можете вносить изменения и улучшения, а затем предложить владельцу исходного репозитория рассмотреть предлагаемые вами корректировки и решить, какие из них действительно заслуживают внимания. Таким образом, вместо попыток получить разрешение на внесение изменений непосредственно в исходный репозиторий вы просто создаете ответвление (fork) (фактически – клон) этого репозитория в своей учетной записи и получаете возможность работы с ним. В дальнейшем, возможно (но не обязательно), владелец исходного ре-

Tlgm: @it_boooks

позитория пожелает принимать не только уже сделанные вами изменения, но и последующие (эта тема рассматривается в следующем разделе «Запросы на включение изменений»).

Для создания ответвления репозитория на GitHub выполняются следующие действия:

1. Регистрация (вход) на сервисе GitHub с использованием требуемых регистрационных данных.
2. Перейти в репозиторий, для которого необходимо создать ответвление в вашей учетной записи, затем щелкнуть по кнопке **Fork** (Ответвление) в верхнем правом углу экрана.
3. Если вы являетесь членом какой-либо организации GitHub, то вам предложат ввести имя пользовательской учетной записи или организации, для которой создается ответвление репозитория. Выберите имя своей пользовательской учетной записи, если остальные варианты вам неизвестны.

Все очень просто. GitHub создаст ответвление (клон) указанного репозитория в вашей учетной записи.

Поскольку репозитории GitHub являются неполными (bare) репозиториями, в большинстве случаев потребуется еще и клонирование нового неполного репозитория на вашу локальную систему, чтобы получить возможность работать с ним. (Отметим, что GitHub предоставляет некоторые инструментальные средства на основе веб-интерфейса для создания и редактирования файлов, для выполнения коммитов и т. п.) Для клонирования репозитория GitHub из вашей учетной записи используется команда `git clone` с указанием URL репозитория GitHub. В качестве примера можно привести URL репозитория одного из авторов данной книги: https://github.com/lowescott/learning-tools.git.

Допустим, что имя вашей учетной записи на GitHub npabook (во время написания книги такого пользователя не было). Если бы вы создавали ответвление указанного выше репозитория, то получили бы абсолютно полную копию этого репозитория в своей учетной записи, как если бы воспользовались командой `git clone`. После создания ответвления URL нового ответвленного репозитория выглядел бы так: https://github.com/npabook/learning-tools.git.

При выполнении команды `git clone https://github.com/npabook/learning-tools.git` из вашей локальной системы Git должен клонировать указанный репозиторий в локальную систему, создать удаленный объект origin, указывающий на ответвленный репозиторий на сервисе GitHub, а также создать удаленные отслеживаемые ветви – точно так же, как это делала команда `git clone` в примерах из предыдущего раздела.

После создания клона репозитория на локальной системе работа с ответвленным репозиторием GitHub выполняется точно так же, как описано в предыдущем разделе:

1. Создаются новые ветви обновлений (feature/topic) в локальной системе, чтобы изолировать вносимые изменения от основной ветви.

Tlgm: @it_boooks

2. Выполняется команда `git push` для внесения этих изменений в удаленный репозиторий GitHub.
3. Выполняется объединение (ввод) изменений с основной ветвью с помощью команды `git merge` в любой момент, когда будет принято решение об их готовности.
4. Выполняется команда `git fetch`, затем команда `git merge` для ввода изменений в основную ветвь. Эти две команды можно заменить одной: `git pull`.
5. Удаляются локальная ветвь обновлений и удаленная отслеживаемая ветвь в репозитории GitHub.

До настоящего момента излагаемый материал не должен был вызывать никаких затруднений у читателей – все описанное в текущем разделе практически ничем не отличалось от содержимого предыдущих разделов. Тем не менее есть тема, требующая пристального внимания, – поддержка синхронизации ответвленного репозитория с исходным.

Поддержка синхронизации ответвленных репозиториев

Несмотря на то что GitHub обеспечивает поддержку связей с исходным репозиторием при создании ответвлений в другой пользовательской учетной записи, сервис не предусматривает какого-либо способа сохранения двух репозиториев в синхронизированном состоянии, а это очень важная функция. Предположим, что вы решили внести определенный вклад в некоторый проект, находящийся в стадии активной разработки. Через некоторое время ваша ответвленная копия безнадежно отстанет от исходного проекта, поскольку его разработка продолжается, объединяются ветви, выполняются коммиты изменений в основную ветвь. Чтобы ответвленный репозиторий стал действительно полезным для внесения изменений, необходима его постоянная поддержка в состоянии, сооответствующем текущему состоянию исходного проекта.

Для сохранения ответвленного репозитория в актуальном состоянии необходимо использовать несколько удаленных объектов (remotes). (Ранее отмечалось, что удаленные объекты достаточно часто используются при работе с Git.) Рассмотрим подробнее, как это должно работать. Предполагается, что вы уже создали ответвленный репозиторий на GitHub.

Во-первых, нужно создать клон ответвленного репозитория на своей локальной системе с помощью команды `git clone`. Команда должна выглядеть следующим образом:

```
git clone https://username@github.com/username/repository-name.git
```

Здесь выполняется клонирование репозитория в локальной системе пользователя, создается удаленный объект origin, указывающий на заданный в команде URL, а также создаются удаленные отслеживаемые ветви. Если в этот момент выполнить команду `git remote` в созданном репозитории, то можно увидеть единственный удаленный объект origin (напомним, что команда `git`

Tlgm: @it_boooks

`remote -v` также выводит место расположения удаленного объекта, в данном примере это HTTPS URL).

Далее добавляется второй удаленный объект, указывающий на исходный репозиторий. Для этого выполняется следующая команда:

```
git remote upstream add https://github.com/original-user/repository-name.git
```

Здесь имя upstream чисто символическое, оно выбрано произвольно. Такое имя полезно своей информативностью – оно напоминает о том, что удаленный объект указывает на проект, находящийся в активной разработке (исходный проект). (Кроме того, можно обнаружить, что имя upstream достаточно часто используется на практике, поэтому определенно имеет смысл применять именно это имя для поддержания логической согласованности.) Теперь в вашем локальном репозитории есть два удаленных объекта: origin, указывающий на ответвленный репозиторий, и upstream, указывающий на исходный репозиторий.

Для поддержки синхронизации вашего локального репозитория с исходным выполняются следующие действия (в клонированном репозитории Git на вашей локальной системе):

1. Переход в основную ветвь с помощью команды `git checkout master`.
2. Получение изменений из исходного репозитория. Можно воспользоваться последовательностью команд `git fetch upstream master` и `git merge upstream/master` или выполнить одну команду `git pull upstream master` (объединяющую функции двух предыдущих команд). После этого локальный клонированный репозиторий будет полностью синхронизирован с исходным репозиторием.
3. Ввод изменений из локального репозитория в ответвленный репозиторий с помощью команды `git push origin master`. После этого ответвленный репозиторий становится полностью синхронизированным с исходным репозиторием.

Описанный выше процесс не обеспечивает синхронизацию всех ветвей обновлений (feature/topic), но, вообще говоря, это не представляет особой проблемы – в основном почти всегда требуется поддержка синхронизации основной ветви исходного и ответвленного репозиториев.

Тема следующего раздела: оповещение владельца исходного репозитория о том, что вы предлагаете изменения, которые он может рассматривать как кандидаты на включение в свой репозиторий.

Запросы на включение изменений

Сначала кратко напомним этапы процесса, предложенного для работы с совместно используемым репозиторием на сервисе GitHub.

1. Создание локальной ветви, обозначаемой как «ветвь обновлений» или «обсуждаемая ветвь», в которой хранятся предполагаемые изменения.
2. Регистрация в индексе и коммит этих изменений в новую локальную ветвь.

Tlgm: @it_boooks

3. Ввод новой локальной ветви в удаленный репозиторий с помощью команды `git push` *`remote branch`*.

При этом изменения передаются в ваш ответвленный репозиторий. Но как владелец исходного репозитория может узнать о том, что вы внесли некоторые изменения в свою ответвленную копию? Если ответить кратко, то никакой возможности нет. Во-первых, вполне возможно, что вы создали действительно полностью ответвленный проект, то есть полностью независимую кодовую базу (codebase), поэтому для авторов исходного проекта нет необходимости знать о вносимых вами изменениях (или они не должны о них знать). Во-вторых, что, если вносимые вами изменения не содержат в полной мере всех обновлений, которые хотели бы рассмотреть авторы исходного проекта? Каким образом Git или GitHub может узнать, готовы ли ваши изменения полностью? Краткий ответ: такой возможности нет. Только вы точно знаете, когда ваш код можно представить на рассмотрение авторов исходного проекта для решения о включении его в разработку. Для решения этой проблемы предназначен запрос на включение изменений.

Запрос на включение изменений (pull request) – это оповещение авторов исходного репозитория о том, что у вас имеются изменения, которые вы предлагаете рассмотреть и принять решение о включении их в исходный репозиторий. Создание запроса на включение изменений выполняется после шага 3 в приведенном выше списке действий. После ввода изменений в ветвь ответвленного репозитория можно создать запрос на включение изменений в исходный репозиторий. (Отметим, что другие платформы на основе Git, такие как GitLab, могут использовать вместо термина «запрос на внесение изменений» (pull request) другие термины, например «запрос на объединение/слияние» (merge request). При этом основная концепция и рабочий процесс остаются практически неизменными.)

Для создания запроса на включение изменений после внесения соответствующей ветви на GitHub необходимо перейти в исходный репозиторий. Непосредственно под строкой, содержащей сведения о коммитах, ветвях, релизах и участниках проекта, появляется новая строка с кнопкой **Compare & pull request** (Сравнение и запрос на включение изменений), как показано на рис. 8.10.

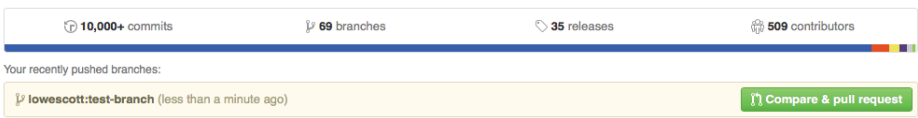


Рис. 8.10 ❖ Создание нового запроса на включение изменений на GitHub

После щелчка по этой кнопке появляется экран создания запроса на включение изменений. Поля, соответствующие базовому ответвлению, базовой ветви, главному ответвлению и сраниваемой ветви, заполнены автоматически,

Tlgm: @it_boooks

а примечание к запросу берется из сообщения самого последнего коммита. Содержимое этих полей можно отредактировать при необходимости, затем щелкнуть по зеленой кнопке **Create pull request** (Создать запрос на включение изменений).

Потом владельцы исходного репозитория должны решить, следует ли включить изменения из вашей ветви обновлений в свой репозиторий. Если они дают положительный ответ (или если вы один из авторов, получивших запрос на включение изменений), то вы получаете возможность внести свои изменения в исходный проект с помощью веб-интерфейса, предоставляемого GitHub.

После включения предложенных изменений в исходный репозиторий можно обновить основную ветвь ответвления из исходного репозитория (используя `git fetch` и `git merge` или одну команду `git pull`). Так как изменения из вашей ветви обновлений теперь находятся в основной ветви, можно удалить ветвь обновлений (а также все удаленные отслеживаемые ветви в ответвленном репозитории), поскольку она больше не нужна.

Очевидно, что, за исключением некоторых незначительных различий общий, рабочий процесс совместного использования сервиса GitHub очень похож на рабочий процесс совместного использования одного (неполного) репозитория. Вообще говоря, в обоих случаях применяются одни и те же термины, концепции и команды. Это существенно упрощает освоение системы Git для совместной работы в группе разработчиков.

Резюме

В этой главе был представлен вводный курс практического использования Git – широко известной системы управления версиями. Git представляет собой полностью распределенную систему управления версиями исходного кода, обеспечивающую полнофункциональную поддержку процесса нелинейной разработки с возможностью создания ветвей. Как и любая другая система управления версиями, Git предлагает механизмы учетных записей (кто имеет право вносить изменения) и отслеживания изменений (информация обо всех внесенных изменениях). Эти атрибуты применимы как при вариантах разработки, ориентированных на сетевую среду, так и при вариантах, ориентированных на разработчика. Ветви являются основным инструментальным средством, обеспечивающим совместную групповую работу с системой Git. Для поддержки совместной групповой работы с Git были созданы онлайновые сервисы (например, GitHub и BitBucket), позволяющие пользователям из различных организаций и регионов организовать коллективную работу в репозиториях без особых затруднений.

Tlgm: @it_boooks

Глава 9

Инструментальные средства автоматизации

При обсуждении автоматизации сети невозможно обойтись без объективной оценки роли инструментальных средств автоматизации – Ansible, Chef, Puppet, StackStorm и Salt.

Обычно эти инструменты принято считать в большей степени предназначенными для автоматизации серверов. Такое мнение можно считать обоснованным, если вспомнить, что большинство этих инструментов (если не все они) появилось в процессе автоматизации и управления серверных и сетевых операционных систем, а также при выполнении процедур конфигурирования сетевых приложений. Но в последние годы многие компании сосредоточили немалые усилия для распространения функциональности автоматизации сети на собственные продукты. В результате эти продукты становятся более удобными и мощными инструментами для сетевой автоматизации в целом.

В этой главе рассматривается практическое использование нескольких наиболее широко известных инструментальных средств автоматизации сети:

- Ansible;
- Salt;
- StackStorm.

Прежде чем перейти к подробностям и примерам применения этих инструментальных средств для автоматизации сети, сделаем краткий обзор изучаемых инструментов.

Краткий обзор инструментальных средств автоматизации

Все названные выше инструменты предназначены для автоматизации, тем не менее каждый инструмент обладает собственной архитектурой, кроме того, реализация процесса автоматизации немного отличается в каждом случае. Таким образом, каждое инструментальное средство имеет свои достоинства и недостатки. В этом разделе кратко описываются все инструментальные средства, для того чтобы сразу стало понятно, как можно применять их в конкретной сетевой среде.

Tlgm: @it_boooks

На самом высоком уровне можно выделить следующие основные различия в архитектуре:

- *использование агентов или отсутствие агентов* – некоторым инструментальным средствам требуются агенты (agents) – небольшие фрагменты программного обеспечения, – выполняемые на управляемой системе или устройстве. В контексте автоматизации сети это может стать проблемой, так как не каждая сетевая операционная система поддерживает функционирование агентов на сетевом устройстве. В тех случаях, когда сетевая ОС не поддерживает собственными средствами работу агента на устройстве, иногда применяются дополнительные способы решения этой проблемы в виде прокси-агента (proxy agent). Очевидно, что инструментальным средствам из противоположной группы агент не требуется, поэтому они могут быть более удобными инструментами автоматизации сети;
- *централизация или децентрализация* – для архитектур, основанных на использовании агентов, часто требуется еще и централизованный «главный сервер» (master server). Некоторые продукты, не использующие агентов, также реализуют концепцию «главного сервера», но большинство инструментальных средств без агентов имеет децентрализованную структуру;
- *специализированный протокол или стандартный протокол* – некоторые инструментальные средства используют собственный специализированный протокол, чаще всего связанный с архитектурами на основе агентов. Другие инструментальные средства в качестве транспортного протокола применяют SSH. С учетом повсеместного распространения протокола SSH на сетевых устройствах инструментальные средства, использующие SSH как транспортный протокол, возможно, лучше подходят для автоматизации сети;
- *предметно-ориентированный язык (DSL) или стандартные форматы данных и языки общего назначения* – некоторые инструментальные средства используют собственный предметно-ориентированный язык. В подобном случае пользователи должны создавать на этом языке соответствующие файлы, обрабатываемые инструментальным средством. DSL – это язык, созданный для решения определенных задач (достижения определенных целей) в конкретной области (или для конкретного инструментального средства). В организациях, пока еще не знакомых с предлагаемым предметно-ориентированным языком, могут возникать дополнительные затруднения (добавляется лишняя «кривая обучения»). Другие инструментальные средства используют YAML, который в данном контексте считается универсальным стандартным языком. Напомним, что язык YAML подробно рассматривался в главе 5;
- *язык для создания расширений* – большинство из вышеназванных инструментальных средств автоматизации поддерживает возможность добавления или расширения функциональности с использованием скрип-

Tlgm: @it_boooks

тового языка высокого уровня. В некоторых инструментах в качестве языка, позволяющего расширить функциональность, выбран Ruby, другие применяют Python;

- *модель «push», модель «pull» или модель, управляемая событиями* – некоторые инструментальные средства формируют свой рабочий процесс на основе модели «push», то есть информация в обязательном порядке передается («проталкивается» – push) из единого центра на все управляемые устройства или системы. Другие инструменты используют модель «pull», в соответствии с которой информация о конфигурации или инструкции обычно передается по запросу («вытягивается» – pull), чаще всего на основе некоторой разновидности планируемого расписания. Кроме того, существуют инструменты, управляемые событиями, которые выполняют определенные действия в ответ на возникновение какого-либо конкретного события или при срабатывании триггера.

Ниже приведен краткий обзор трех инструментальных средств, рассматриваемых в текущей главе, с учетом всех вышеописанных архитектурных различий:

- *Ansible* – имеет децентрализованную архитектуру без применения агентов и использует SSH как базовый транспортный протокол. Обычно работает на основе модели push, но поддерживает и модель pull. Инструментальное средство Ansible написано на языке Python и использует этот язык для расширения функциональности. Ansible поддерживает работу с шаблонами, написанными на языке Jinja. Инструмент Ansible изначально позиционировался как средство оперативного выполнения специализированных команд на серверах, но со временем развился в мощное средство оркестровки задач с использованием так называемых «комплектов сценариев» (playbooks), которые выполняют типовые задачи с неизменным результатом на целевых системах. Сценарные книги могут быть написаны на стандартном языке YAML или на диалекте языка YAML, специализированном для Ansible;
- *Salt* – обладает гибкой архитектурой, в которой возможно как использование агентов, так и отказ от них. В варианте архитектуры с агентами Salt-агенты обмениваются информацией с главным сервером Salt через шину сообщений (message bus). В варианте архитектуры без агентов обмен информацией ведется на основе протокола SSH или с помощью других библиотек от независимых производителей, например NAPALM (будет рассматриваться ниже). Инструментальное средство Salt написано на языке Python и использует этот язык для расширения функциональности. По умолчанию язык Jinja обеспечивает функциональность шаблонов. Salt появился как инструмент для удаленного управления сервером и во многом похож на Ansible, но со временем вырос в надежное и мощное средство управления конфигурацией через механизмы Salt States, написанные на языке YAML. У Salt есть одна особенность – этот инструмент также является платформой для реализации управляемой

Tlgm: @it_boooks

событиями автоматизации, а не только средством управления общей сетевой конфигурацией;

- *StackStorm* – применяет абсолютно другой подход, по сравнению с перечисленными выше инструментальными средствами. StackStorm сосредоточен исключительно на автоматизации, управляемой событиями, то есть задачи выполняются в ответ на происходящие события. StackStorm использует язык Python для создания сенсоров, генерирующих события, и акторов, выполняющих задачи. В StackStorm применяется язык YAML в некоторых особых случаях для предоставления метаданных сенсорам и акторам или для определения рабочего процесса (потока).

После приведенного выше краткого обзора можно перейти к более подробному рассмотрению каждого инструментального средства и способов их практического применения для автоматизации сети. Инструменты будут рассматриваться в алфавитном порядке, поэтому начнем с Ansible.

Использование Ansible

Почти все инструментальные средства, рассматриваемые в текущей главе, требуют нескольких месяцев (как минимум) для освоения их на профессиональном уровне. Цель данной главы и в частности данного раздела, посвященного Ansible, – предоставить читателю достаточный объем информации, необходимой для быстрого начала работы и немедленного выполнения наиболее насущных задач. Для этого в разделе выделено шесть основных тем (подразделов):

- основы работы Ansible;
- создание inventory-файла;
- выполнение сценария Ansible;
- предназначение различных файлов;
- создание сценариев Ansible для автоматизации сети;
- использование сторонних модулей.

Внимательно изучив этот раздел, вы получите полную информацию об основах практического использования Ansible, о различных типах автоматизации сети, которые можно реализовать с помощью Ansible, а также, что более важно, информацию, необходимую для продолжения углубленного изучения процесса автоматизации сети.

Начнем с изучения основ работы с Ansible.



Ansible – это платформа с открытым исходным кодом, созданная компанией Red Hat. В этом разделе мы рассматриваем только ядро Ansible – собственно, саму платформу с открытым исходным кодом. Но следует отметить, что у Red Hat имеется и коммерческое предложение – Ansible Tower на основе ядра Ansible, в котором предоставляются функциональные возможности корпоративного уровня, такие как система управления доступом Role Based Access Control (RBAC), защищенное хранилище для сетевых регистрационных данных, а также ряд других функций, например RESTful API, так что есть возможность программно управлять и выполнять комплекты сценариев (playbook) Ansible (это далеко не полный список предлагаемых функций).

Tlgm: @it_boooks

Основы работы Ansible

При изучении основ работы Ansible сразу следует отметить, что с точки зрения архитектуры процесс автоматизации реализован не только для сетевых устройств, но и для серверов Linux. Это различие может показаться малозначимым, но оно очень важное, поскольку наиболее рельефно проявляется при создании рабочего процесса (потока) с помощью Ansible.

Автоматизация серверов Linux

При использовании Ansible для автоматизации серверов Linux этот инструмент работает как распределенная программная среда. Ansible управляет хостом, то есть компьютером, на котором установлено программное обеспечение Ansible, и устанавливает соединения по протоколу SSH со всеми автоматизируемыми серверами. Управляющий хост последовательно копирует код на языке Python на каждый сервер – этот код, собственно, и выполняет задачу автоматизации. Процесс включает все действия: от перезапуска процессов ОС Linux и установки требуемых пакетов Linux до обновления текстовых (конфигурационных) файлов, получения обновлений из репозитория Git или простого запуска скриптов командной оболочки bash на автоматизируемых хостах (серверах) Linux. Если необходимо автоматизировать 100 серверов Linux, то на всех 100 серверах выполняется «код» (или модули) Ansible для решения задачи автоматизации.



Ansible можно также использовать для автоматизации серверов Windows, но тематика текущего раздела предполагает рассмотрение основ и наиболее общих вариантов практического использования Ansible, а также сравнение процессов автоматизации серверов и сетевых устройств.

Автоматизация сетевых устройств

При автоматизации сетевых устройств Ansible работает немного по-другому – как локальная или централизованная программная среда. В этом случае Ansible управляет конфигурируемым хостом в локальном режиме. При работе в локальном режиме Ansible не устанавливает соединения с каждым устройством по протоколу SSH и не копирует код Python на каждое устройство. В действительности Ansible устанавливает соединение с самим собой и выполняет код Python локально. Код на языке Python, выполняемый локально, тем не менее может устанавливать соединение с сетевым устройством по протоколу SSH, но также возможна работа через API (или по протоколам Telnet или SNMP). Но даже если Ansible устанавливает соединение по протоколу SSH с сетевыми устройствами, файлы с кодом Python не копируются на устройство, а через SSH-соединение просто передаются команды CLI. Если рассматривать процесс на самом высоком уровне, то можно провести аналогию с выполнением скриптов на языке Python на сервере и автоматизацией нескольких сетевых устройств в параллельном режиме.

Tlgm: @it_boooks



Возможно наличие сетевых модулей, которые не работают в локальном режиме, но при этом требуются обновления многих экземпляров сетевых операционных систем. Для поддержки такого режима необходимо разрешение оперативного установления SSH-соединения с каждым сетевым устройством с последующим копированием файлов с кодом Python во временный каталог и выполнение этих файлов с помощью соответствующей среды Python. Разумеется, полная поддержка этого процесса обеспечена в среде Cumulus Linux. В настоящее время этот процесс поддерживают еще две операционные системы: Cisco IOS-XR и Arista EOS.

После изучения различий в работе Ansible на сетевых устройствах, по сравнению с работой на хостах Linux (для которых изначально было создано инструментальное средство Ansible), можно перейти к рассмотрению одной особенности Ansible, которая чрезвычайно важна при освоении этого инструмента.

Создание inventory-файла

Изучение практического использования Ansible следует начать с inventory-файла. Inventory-файл – это один из двух файлов, необходимых для начала процесса автоматизации сетевых устройств с применением Ansible. Второй файл, который мы рассмотрим немного позже, – это сценарий (playbook). Inventory-файл в определенной степени похож на ini-файл, содержащий список устройств, которые должны быть автоматизированы с помощью Ansible.

Ниже приведен пример inventory-файла Ansible:

```
10.1.100.10
10.5.10.10
nyc-lf01
```

В примере показан один из самых простых inventory-файлов. Файл содержит три строки, по одной для каждого устройства. Здесь можно видеть, что по умолчанию можно использовать IP-адреса или символьные имена хостов (которые должны быть полностью определенными именами доменов).

На первый взгляд inventory-файл может показаться очень простым, но в него можно добавлять структуры и данные в процессе освоения работы с Ansible. Если вы начинаете работу с тремя устройствами и требуется лишь тестирование, то приведенного выше примера вполне достаточно. Но в более реальных сценариях приходится учитывать наличие большего количества устройств разнообразных типов и необходимость развертывания этих устройств в различных частях сети (например, в центре данных, DMZ, WAN, в сегменте с прямым доступом и т. д.). В следующих примерах будет показано создание групп и определение переменных в inventory-файле.

Рассмотрим подробнее создание inventory-файла, в котором представлены устройства для двух регионов *EMEA* и *AMERS* и две роли *dc* и *cpe* (см. рис. 9.1). Каждая роль устройства в отдельном регионе содержит различные типы сетевого оборудования, как показано на схеме.

Tlgm: @it_boooks

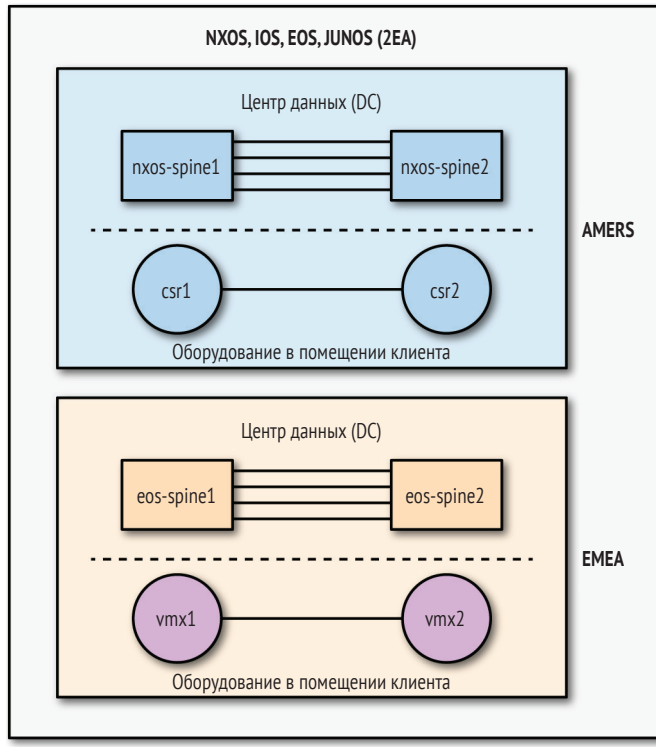


Рис. 9.1 ❖ Схема топологии сети

Устройства, показанные на рис. 9.1, используются в разделах текущей главы, в которых рассматриваются Ansible и Salt. В регионе AMERS маршрутизаторы Cisco используются как граничные устройства в комплекте оборудования в помещении клиента (CPE), а коммутаторы Nexus – в центре данных (DC). В регионе EMEA маршрутизаторы Juniper используются как граничные устройства в комплекте оборудования в помещении клиента (CPE), а коммутаторы Arista – в центре данных (DC).

Работа с группами inventory-файла

На схеме можно видеть, что существуют две географические локации в регионах AMERS и EMEA с различными типами устройств. Необходимо создать различные группы, в которых можно будет провести процедуру автоматизации для всех устройств определенного типа, или для конкретной операционной системы, или в границах одной из локаций.

Для решения этой задачи начнем с создания в регионе AMERS двух групп для устройств с именами `amers-cpe` и `amers-dc`. В каждую группу включены два устройства. Группа `cpe` содержит два устройства Cisco CSR 1000V, в группе `dc` – два коммутатора Nexus.

Tlgm: @it_boooks


```
[amers-cpe]
csr1
csr2

[amers-dc]
nxos-spine1
nxos-spine2
```

Синтаксическая конструкция с применением квадратных скобок [] предоставляет возможность создания логических групп в inventory-файле. Немного позже вы увидите, как можно ссылаться на эти группы в сценарии для упрощения процедуры автоматизации.

Использование вложенных групп в inventory-файле

В inventory-файле можно также создавать группы групп или вложенные группы. Например, при выполнении процедуры автоматизации всех устройств в регионе AMERS. Для этого нужно добавить группу с именем `amers`, которая, в свою очередь, содержит группы `amers-cpe` и `amers-dc`.

```
[amers:children]
amers-cpe
amers-dc

[amers-cpe]
csr1
csr2

[amers-dc]
nxos-spine1
nxos-spine2
```

При создании вложенных групп необходимо обязательно использовать суффикс `:children` в определении имени родительской группы. После добавления группы `amers` в рассматриваемом inventory-файле определены три группы: `mers`, `amers-cpe` и `amers-dc`.

Воспользовавшись этой методикой, можно сформировать аналогичный inventory-файл для региона EMEA, как показано в следующем примере:

```
[emea:children]
emea-cpe
emea-dc

[emea-cpe]
vmx1
vmx2

[emea-dc]
eos-spine1
eos-spine2
```

Кроме того, можно создавать группы для автоматизации всех устройств с конкретно определенной ролью в сети, например для всех устройств, разме-

Tlgm: @it_boooks

щенных в помещении клиента (CPE), или для всех устройств в центре данных (DC). Для этого можно создать дополнительные группы.

```
[all-cpe:children]
amers-cpe
emea-cpe

[all-dc:children]
amers-dc
emea-dc
```

В приведенных выше примерах можно видеть, что изначально создаваемый inventory-файл может быть чрезвычайно простым, но в процессе автоматизации сетевых устройств этот файл быстро расширяется и дополняется.

После изучения групп, определяемых пользователем, и создания из них требуемой структуры можно перейти к рассмотрению возможности определения переменных в inventory-файле.

Использование переменных в Ansible

В inventory-файле Ansible можно определять следующие два типа переменных:

- переменные группы;
- переменные хоста.

Использование переменных группы Переменные группы (group variables) определяются на уровне группы. Например, можно определить переменную, содержащую IP-адрес сервера NTP для использования всеми устройствами региона AMERS, и аналогичную переменную с IP-адресом NTP для устройств в регионе EMEA.

Определения переменных в inventory-файле выглядят следующим образом:

```
[amers:vars]
ntp_server=10.1.200.11

[emea:vars]
ntp_server=10.10.200.11
```

Для определения переменных группы в inventory-файле создается новый раздел с именем соответствующей группы и суффиксом `:vars`. В этом разделе определяются переменные уровня данной группы. В приведенном примере для каждой группы создана одна переменная с именем `ntp_server`. Если в процессе автоматизации устройств в регионе AMERS выполняется обращение к этой переменной, то используется адрес `10.1.200.11`. Если автоматизация осуществляется для группы EMEA, то используется значение `10.10.200.11`.



К порядку размещения групп и переменных в inventory-файле не предъявляется каких-либо особых требований. Можно сначала определить все группы, затем все переменные или помещать определения переменных после определения соответствующей группы, то есть группа1, переменные группы1, группа2, переменные группы2 и т. д.

Использование переменных хоста В inventory-файл также можно определять переменные уровня хоста, предназначенные для конкретного устройства.

Tlgm: @it_boooks

Определение переменной хоста размещается в той же строке, что и имя хоста. Выше приводился пример определения переменной с адресом выделенного сервиса NTP для каждого региона. Но что делать, если устройство `nxos-spine1` обязательно должно использовать другой адрес?

Проблема легко решается следующим образом:

```
[amers-dc]
nxos-spine1  ntp_server=10.1.200.200
nxos-spine2
```

Добавление конструкции `variable=value` в строку с именем устройства представляет собой способ определения переменной уровня хоста в inventory-файле. В одной строке может быть определено несколько переменных хоста, как показано в следующем примере:

```
[amers-dc]
nxos-spine1  ntp_server=10.1.200.200  syslog_server=10.1.200.201
nxos-spine2
```

Определение приоритета переменных Важно понимать, что более специализированные переменные имеют более высокий приоритет, то есть при наличии группы `amers` и вложенной в нее группы `amers-cpe` переменная группы `amers-cpe` замещает переменную с таким же именем, определенную для группы `amers`. Если существует переменная хоста для устройства из группы `amers-cpe`, то она имеет еще более высокий приоритет и замещает переменную группы `amers-cpe`.

Существует множество способов использования переменных, которые в нашей книге не рассматриваются, поэтому приоритет переменных, не создающий никаких проблем в самых простых случаях (на начальном этапе обучения), может стать весьма важной темой, заслуживающей особого внимания в крупных проектах Ansible. Опеределение приоритета переменных подробно описано на веб-сайте Ansible.

Группа all Следует знать, что всегда существует неявно определенная группа с именем `all`. Эта группа используется для автоматизации всех устройств, содержащихся в inventory-файле. Для группы `all` также можно определять переменные соответствующего уровня, как показано ниже:

```
[all:vars]
ntp_server=10.1.200.199
syslog_server=10.1.200.201
```

Переменные, определенные для группы `all`, становятся «переменными по умолчанию» и используются только в тех случаях, когда переменная с аналогичным именем не определена для более специализированной группы или как переменная хоста.



Необходимо хорошо понимать, как работают переменные уровня группы и переменные хоста, но inventory-файл – это не самое подходящее место для определения всех переменных для крупного проекта. Для определения переменных рекомендуется использовать специально выделенные файлы, которые рассматривались выше в подразделе «Использование переменных хоста».

Tlgm: @it_boooks

Если в процессе чтения предыдущих разделов вы выполняли и тестировали все примеры, то сейчас необходимо добавить еще несколько групп и переменных в созданный inventory-файл, как показано ниже:

```
[nxos]
nxos-spine1
nxos-spine2

[nxos:vars]
os=nxos

[eos]
eos-spine1
eos-spine2

[eos:vars]
os=eos

[iosxe]
csr1
csr2

[iosxe:vars]
os=ios

[junos]
vmx1
vmx2

[junos:vars]
os=junos
```

Причина добавления групп для каждой ОС и соответствующих переменных для них заключается в том, что в большинстве случаев такая информация требуется для автоматизации указанного устройства, так как многие модули для этого устройства либо работают только в конкретных операционных системах, либо предоставляются различными производителями сетевого оборудования и требуют явного определения используемой ОС.

После завершения формирования полный inventory-файл приобретает следующий вид:

```
[all:vars]
ntp_server=10.1.200.199
syslog_server=10.1.200.201

[amers-cpe]
csr1
csr2

[amers-dc]
nxos-spine1
nxos-spine2

[amers:children]
amers-cpe
amers-dc
```

Tlgm: @it_boooks


```
[amers-cpe]
csr1
csr2

[amers-dc]
nxos-spine1  ntp_server=10.1.200.200  syslog_server=10.1.200.201
nxos-spine2

[emea:children]
emea-cpe
emea-dc

[emea-cpe]
vmx1
vmx2

[emea-dc]
eos-spine1
eos-spine2

[all-cpe:children]
amers-cpe
emea-cpe

[all-dc:children]
amers-dc
emea-dc

[amers:vars]
ntp_server=10.1.200.11

[emea:vars]
ntp_server=10.10.200.11

[nxos]
nxos-spine1
nxos-spine2

[nxos:vars]
os=nxos

[eos]
eos-spine1
eos-spine2

[eos:vars]
os=eos

[iosxe]
csr1
csr2

[iosxe:vars]
os=ios

[junos]
vmx1
vmx2

[junos:vars]
os=junos
```

Tlgm: @it_boooks



Самый простой и наиболее распространенный способ освоения работы с inventory Ansible – это практическое использование inventory-файла. Но если вам приходится работать в динамически изменяющейся или весьма крупной сетевой среде либо пользоваться существующей базой данных CMDB или системой управления сетью, содержащей inventory-данные, есть возможность интеграции Ansible в такие системы. Ansible поддерживает динамические inventory-скрипты, то есть inventory-файл заменяется на скрипт. Этот скрипт обращается с запросом к БД CMDB/NMS, выполняет нормализацию данных, затем возвращает требуемые данные в корректном формате JSON в структуре Ansible, соответствующей требованиям, определенным в документации (эта тема не рассматривается в нашей книге). Кроме того, переменные могут возвращаться в динамическом inventory-скрипте, и если вы работаете в крупной организации, то все, что вам нужно, – управлять комплектами сценариев (playbooks), а переменные будут возвращаться динамически при выполнении соответствующего сценария.

В начале раздела было отмечено, что inventory-файл является одним из двух файлов, требуемых для начала процесса автоматизации сетевых устройств с использованием Ansible. В следующем разделе рассматривается второй необходимый файл – комплект сценариев Ansible.

Выполнение сценария Ansible

Комплект сценариев (playbook) – это файл, который содержит инструкции по автоматизации. Иначе говоря, каждый сценарий содержит отдельные задачи и рабочие потоки (процессы), которые вы предполагаете использовать для автоматизации сети. Сценарий пишется на языке YAML. Поэтому нам снова потребуется язык YAML, который рассматривался в главе 5.

Английский термин *playbook* взят из области спорта, его значение – «план, схема игры» (другое значение слова playbook – сборник сценариев или пьес). Таким образом, каждый файл комплекта сценариев представляет собой общий «план» (или «сборник»), содержащий один или несколько сценариев (plays). В то же время каждый сценарий может содержать одну или несколько задач (tasks).

Приведем конкретный пример комплекта сценариев, чтобы лучше понять его структуру и связанную с ним терминологию. Мы рассмотрим сценарии (plays), задачи (tasks), модули (modules), а также использование переменных в комплекте сценариев. Кроме того, будет объяснена связь между комплектом сценариев и inventory-файлом.

```
---
  - name: PLAY 1 - ISSUE SNMP COMMANDS
    hosts: iosxe
    connection: local
    gather_facts: no

    tasks:

      - name: TASK1 - DEPLOY SNMP COMMANDS
        ios_command:
          commands:
```

Tlgm: @it_boooks


```
      - show run | inc snmp
    provider:
      username: ntc
      password: ntc123
      host: "{{ inventory_hostname }}"

- name: TASK 2- DEPLOY SNMP COMMANDS
  ios_config:
    commands:
      - snmp-server community public RO
    provider:
      username: ntc
      password: ntc123
      host: "{{ inventory_hostname }}"
```

Комплект сценариев из этого примера содержит один сценарий и две задачи. В главе 5 вы ознакомились с основами языка YAML и правилами форматирования строк для этого языка. Обязательным требованием является наличие списка сценариев в формате YAML и списка задач (также в формате YAML) как значения ключа `tasks`. Если форматирование (отступы) строк нарушено даже незначительно, то при попытке выполнения такого комплекта сценариев Ansible выводит сообщения об ошибках, поэтому будьте внимательны.

Приступим к подробному разбору содержимого файла комплекта сценариев:

- файлы на языке YAML всегда начинаются с трех дефисов `---`, а поскольку комплект сценариев пишется на языке YAML, необходимо соблюдать это правило;
- следующие несколько строк – определение первого (и единственного) сценария. Обратите внимание на ключи `name`, `hosts`, `connection` и `gather_facts`;
  - `name` – это не обязательный, но рекомендуемый ключ. Он используется для краткого определения целей сценария и выполняемых им задач. Иначе говоря, это произвольный текст;
  - `hosts` – ключ определяет, какие устройства необходимо автоматизировать. Здесь можно указать хост, группу, несколько хостов или групп или даже выражения, ссылающиеся на имена в inventory-файле. Например, если нужно автоматизировать все устройства в регионе EMEA, используя inventory-файл из предыдущего раздела, то можно определить этот ключ как `hosts: emea`. Для устройств из групп IOS и NXOS определение будет выглядеть так: `hosts: iosxe, nxos`. Группы в списке отделяются друг от друга запятой;
  - `connection` – ключ определяет тип соединения, используемый сценарием. Для большинства сетевых устройств требуется тип соединения `local`. Это полностью соответствует архитектуре Ansible, которая рассматривалась выше в разделе «Автоматизация сетевых устройств»;
  - `gather_facts` – поскольку процесс автоматизации сетевых устройств и сопутствующие процедуры выполняются в локальном режиме, сбор

Tlgm: @it_boooks

фактов на компьютере, на котором работает Ansible, не производится (задано значение no). Если бы система Ansible работала в режиме автоматизации серверов Linux, то необходимо было бы по умолчанию определить функцию сбора фактов от каждого узла, включая такие элементы, как тип ОС, версия ОС, производитель, IPv4-адреса и многое другое.

Теперь рассмотрим подробнее, что представляют собой задачи и как они связаны с модулями.

Задачи в сценариях Ansible

После определения атрибутов (характеристик) самого высокого уровня для сценария, таких как тип соединения и автоматизируемые хосты, необходимо определить задачи. Каждая задача (task) запускает модуль (module) Ansible, который выполняет некоторую форму (вариант) автоматизации. В приведенном выше примере две задачи, каждая из которых обозначена собственным именем.

Как и при определении сценария, общепринятой практикой считается использование ключа name при определении каждой задачи. Это позволяет точно описать, что именно происходит при активизации задачи в реальном времени при выполнении данного комплекта сценариев.

Модули Ansible

На уровне форматирования (отступа) строки name в обеих задачах можно видеть строки ios_command (в первой задаче) и ios_config (во второй задаче). Это *модули* (*modules*) Ansible, которые выполняют специализированные операции. В частности, ios-command используется для выполнения команд exec-уровня в устройствах Cisco IOS, а ios_config применяется для передачи команд конфигурационного уровня в устройства Cisco IOS.

В этом разделе будут рассматриваться различные модули в следующих примерах комплектов сценариев.

В Ansible имеется более 700 модулей, позволяющих автоматизировать серверы Linux, сетевые устройства, серверы Windows, общедоступные облачные среды и многое другое.

Еще раз обратимся к задаче, которая использует модуль ios_command:

```
- name: TASK1 - DEPLOY SNMP COMMANDS
  ios_command:
    commands:
      - show run | inc snmp
    provider:
      username: ntc
      password: ntc123
      host: "{{ inventory_hostname }}"
```

Отметим, что под строкой ios_command размещены дополнительные данные (также отформатированные соответствующим образом), обозначенные как

Tlgm: @it_boooks

commands: и provider:. Эти обозначения представляют собой параметры, передаваемые в модуль. Это можно считать аналогом передачи переменных или пар ключ-значение в функцию языка Python.

При использовании Ansible чрезвычайно важно хорошо понимать типы данных языка YAML. Например, параметр commands должен содержать список (list), а параметр provider должен содержать словарь (dictionary). Оба этих типа данных подробно рассматривались в главах 4 и 5.

Для сетевого инженера передача списка команд на сетевое устройство, вероятнее всего, не вызывает никаких вопросов, поэтому сосредоточимся на втором параметре. Параметр provider – это словарь, используемый для передачи информации о соединении, как, например, регистрационные данные, автоматизируемый хост (устройство), информация о номере порта и протоколе. В приведенном примере параметр provider содержит три пары ключ-значение: username, password и host. Примечание: на самом верхнем уровне параметр provider поддерживает многие другие ключи на основе используемых ключей SSH, настраиваемых паролей и изменяемой информации о номере порта и о протоколе.

В рассматриваемом примере значения, передаваемые в ключах username и password, представляют собой обычный текст для наглядной демонстрации взаимосвязи между задачами, модулями и параметрами. Последнему ключу host присвоено значение {{ inventory_hostname }}. Это значение характеризуется двумя фактами, которые необходимо знать.

Во-первых, требуется понимание синтаксиса этой переменной в Ansible. Если говорить более точно, это переменная языка шаблонов Jinja. Форма переменной должна быть вам знакома, поскольку она подробно описывалась при рассмотрении шаблонов конфигурации Jinja в главе 6. Конкретная переменная inventory_hostname – это встроенная переменная Ansible, соответствующая имени устройства, определенному в inventory-файле. В рассматриваемом примере комплекта сценариев автоматизируются все устройства в группе iosxe, состоящей из двух маршрутизаторов Cisco. Для всех устройств группы iosxe выполняются обе указанные задачи Ansible. Если автоматизируемое устройство csr1, то значением inventory_hostname становится csr1, если же автоматизируемое устройство csr2, то inventory_hostname получает значение csr2.

Мы выполнили краткий обзор общего устройства комплекта сценариев на высоком уровне. Теперь необходимо более подробно рассмотреть, как работают конкретные модули, в том числе модули ios_command и ios_config, а также узнать намного больше о системе Ansible и об автоматизации сетевых устройств.

Выполнение комплекта сценариев

Как утверждалось выше, для начала работы с Ansible требуются два файла: inventory-файл и комплект сценариев. В предыдущих разделах мы рассмотрели оба этих файла и теперь готовы к выполнению комплекта сценариев для набора сетевых устройств.

Tlgm: @it_boooks



В рассматриваемом примере inventory-файл был сохранен под именем inventory, а комплект сценариев – под именем snmp-intro.yml. Оба имени выбраны произвольно, пользователь может выбирать любые имена.

Для выполнения комплекта сценариев используется программа ОС Linux `ansible-playbook` с флагом `-i` для ссылки на inventory-файл и указанием имени файла комплекта сценариев:

```
$ ansible-playbook -i inventory snmp-intro.yml

PLAY [PLAY 1 - ISSUE SNMP COMMANDS] ****************************************

TASK [TASK1 - DEPLOY SNMP COMMANDS] ****************************************
ok: [csr2]
ok: [csr1]

TASK [TASK 2- DEPLOY SNMP COMMANDS] ****************************************
ok: [csr2]
ok: [csr1]

PLAY RECAP *****************************************************
csr1                       : ok=2    changed=0    unreachable=0    failed=0
csr2                       : ok=2    changed=0    unreachable=0    failed=0
```

Можно избавиться от необходимости указывать inventory-файл при каждом выполнении комплекта сценариев. Для этого воспользоваться inventory-файлом, определенным по умолчанию, – */etc/ansible/hosts*, присвоив это значение переменной среды `ANSIBLE_INVENTORY`, или явно определить inventory в файле конфигурации *ansible.cfg*. Например, это можно сделать следующим образом:

```
$ export ANSIBLE_INVENTORY=inventory
$
$ ansible-playbook snmp-intro.yml
$
```

Использование файлов переменных

Если вы хорошо понимаете, как формируется inventory-файл с использованием групп и переменных, и освоили основы написания комплектов сценариев, то уже сможете кое-что сделать с помощью Ansible. Но в inventory-файле не рекомендуется размещать большое количество переменных. Для тестирования такой подход удобен и прост, но при управлении крупной системой предприятия необходимо использовать файлы переменных (если не используется механизм dynamic inventory и CMDB (БД управления конфигурацией)).

Использование файлов переменных требует знания и понимания правил именования переменных. Общая концепция применения переменных практически не отличается от методики применения переменных уровня групп и хостов в inventory-файле. Единственное различие состоит лишь в том, что переменные хранятся в отдельном YAML-файле.

Tlgm: @it_boooks

Файлы переменных уровня группы

Чтобы поместить переменные группы в YAML-файлы, необходимо сохранить их в файле в каталоге *group_vars*. Часто в этом же каталоге размещаются и файлы комплектов сценариев для простых проектов. Это особенное и единственное в своем роде имя для Ansible, поэтому каталог обязательно должен называться *group_vars*.

Существуют два варианта работы с каталогом *group_vars*. Первый и наиболее часто используемый начинающими осваивать Ansible состоит в том, что имена YAML-файлов соответствуют именам групп, определенных в inventory-файле. При формировании inventory-файла в нашем примере были явно определены группы `emea`, `amers` и `iosxe`, но, кроме того, существует еще и неявно определенная группа `all`. Для определения переменных уровня этих групп необходимы файлы с именами *emea.yml*, *amers.yml*, *iosxe.yml* и *all.yml*. Такие файлы требуются только в том случае, если требуется определение переменных для конкретных групп.

Ниже приведен пример определения переменных для группы `amers`:

```
ntc@ntc:~/testing$ more group_vars/amers.yml
---

snmp:
  contact: Joe Smith
  location: AMERICAS-NJ
  communities:
    - community: public
      type: ro
    - community: public123
      type: ro
    - community: private
      type: rw
    - community: secure
      type: rw
ntc@ntc:~/testing$
```

Аналогичные файлы переменных создаются для каждой группы при необходимости.

По мере роста количества переменных в какой-либо группе более логичным может показаться размещение переменных в нескольких файлах, соответствующих определенным подгруппам, например переменные AAA в одном файле, переменные NTP в другом и т. д. Это второй вариант.

Во втором варианте создается подкаталог с именем соответствующей группы, и в нем размещаются файлы с произвольными именами, выбираемыми пользователем. Ниже приведен пример, демонстрирующий оба варианта размещения переменных:

```
ntc@ntc:~/testing/group_vars$ tree
.
├── all.yml
```

Tlgm: @it_boooks


```
├── amers.yml
├── apac
│   ├── aaa.yml
│   ├── interfaces.yml
│   └── ntp.yml
└── emea.yml

1 directory, 6 files
```

Файлы переменных уровня хоста

Использование файлов переменных уровня хоста ничем не отличается от использования файлов переменных групп, но каталог для хранения файлов называется host_vars, а файлы (или подкаталоги) должны соответствовать именам устройств, определенным в inventory-файле.

Ниже приведен пример, демонстрирующий два варианта размещения файлов переменных, аналогичный примеру для переменных групп из предыдущего раздела:

```
cisco@cisco:~/testing/host_vars$ tree
.
├── csr1.yml
├── csr2.yml
└── vmx1
    ├── interfaces.yml
    └── ntp.yml

1 directory, 4 files
```

После рассмотрения переменных и способов их правильного использования мы переходим к созданию более полезных комплектов сценариев Ansible.

Создание комплектов сценариев Ansible для автоматизации сети

К настоящему моменту мы сделали краткий обзор архитектуры Ansible, подробно рассмотрели inventory-файлы и комплекты сценариев Ansible, а также познакомились с общими терминами, такими как сценарий, задача, модуль, параметр и переменная. В этом разделе рассматриваются различные модули и особые функциональные характеристики Ansible с уделением повышенного внимания к возможностям этого инструмента в области автоматизации сети. Мы сосредоточимся на конкретных способах и методиках использования Ansible для автоматизации следующих задач:

- создание шаблонов конфигурации для устройств от различных производителей и автоматическая генерация конфигураций;
- развертывание конфигураций и проверка существования заданных конфигураций;
- сбор данных, принимаемых от сетевых устройств;
- выполнение процедур проверки согласованности;
- генерация отчетов.

Tlgm: @it_boooks

Типы основных сетевых модулей

Прежде чем приступить к изучению способов и методик решения вышеперечисленных задач на практических примерах, необходимо рассмотреть полный набор основных модулей, которые Ansible предлагает для работы с устройствами многочисленных производителей сетевого оборудования и различными операционными системами. Это чрезвычайно важно, поскольку рассматриваемые здесь модули представляют собой ядро подсистемы модулей, входящее в комплект поставки Ansible, и все эти модули функционируют в одинаковой манере. В ядре подсистемы модулей Ansible существуют три типа модулей:

- `command` – используются для передачи команд exec-уровня на сетевые устройства. Модули этого типа имеют имена вида *x*`os_command` (например, `ios_command`, `nxos_command`, `junos_command` и .т. д.) в соответствии с конкретной реализацией;
- `config` – используются для передачи конфигурационных команд на сетевые устройства. Модули этого типа имеют имена вида *x*`os_config` (например, `ios_config`, `nxos_config`, `junos_config` и т. д.) в соответствии с конкретной реализацией;
- `facts` – используются для сбора информации, получаемой от сетевых устройств, например версия ОС, аппаратная платформа, серийный номер, имя хоста, соседние узлы/устройства и т. п. Модули этого типа имеют имена вида *x*`os_facts` (например, `ios_facts`, `nxos_facts`, `junos_facts` и т. д.) в соответствии с конкретной реализацией.

С этими тремя типами модулей для каждой сетевой ОС можно выполнить некоторый объем работы по автоматизации сети с использованием Ansible. Для каждого из модулей существуют специальные параметры, такие как `commands` и `provider`, которые уже встречались в предыдущих разделах, и многие другие. Некоторые параметры будут рассматриваться при разборе примеров текущего раздела.



Чтобы узнать, какие параметры поддерживает тот или иной модуль, а также увидеть практические примеры их применения, можно воспользоваться утилитой `ansible-doc`. Например, чтобы понять, как нужно работать с модулем `ios_config`, в командной строке оболочки bash можно выполнить следующую команду:

```
$ ansible-doc ios_config
```

После краткого ознакомления с основными модулями ядра Ansible можно перейти к рассмотрению первого практического примера выполнения одной из задач автоматизации сети.

Создание и использование шаблонов конфигурации

Цель первого примера – показать, как можно использовать Ansible для автоматической генерации конфигураций SNMP при предполагаемом наличии шаблонов конфигурации SNMP и соответствующих входных данных.

Ниже приведен список команд SNMP, выполняемых из командной строки и необходимых для развертывания IOS, но основной целью является развер-

Tlgm: @it_boooks

тывание точно таких же конфигурационных данных в операционных системах Arista EOS, Cisco NXOS и Juniper Junos. Рассмотрим, как можно выполнить эту задачу, создавая проект Ansible.

```
snmp-server location AMERICAS-NJ
snmp-server contact Joe Smith
snmp-server community public RO
snmp-server community public123 RO
snmp-server community private RW
snmp-server community secure RW
```

Сначала необходимо преобразовать приведенные выше команды в шаблоны конфигурации с использованием Jinja и YAML-файлов, в которых будут храниться требуемые входные данные как переменные, точнее говоря, это будут файлы переменных уровня группы.

Так как поставлена задача поддержки четырех операционных систем, потребуются шаблоны на языке Jinja для каждой сетевой ОС. Также необходимо учесть, что входные данные могут отличаться для разных географических регионов, и это будет отображено в нашем примере. Кроме того, будет продемонстрировано использование одного и того же шаблона с различными наборами данных, поскольку для каждого региона существуют собственные строки сообщества, контактные данные и локация в соответствии с протоколом SNMP.

Создание файлов переменных Сначала выведем значения из каждого файла переменных, соответствующего созданным ранее группам. Первыми выводятся данные SNMP, используемые для устройств в американском регионе:

```
ntc@ntc:~/testing$ more group_vars/amers.yml
---

snmp:
  contact: Joe Smith
  location: AMERICAS-NJ
  communities:
    - community: public
      type: ro
    - community: public123
      type: ro
    - community: private
      type: rw
    - community: secure
      type: rw
ntc@ntc:~/testing$
```

Далее выводится аналогичная структура данных для региона EMEA.

```
$ more group_vars/emea.yml
---

snmp:
  contact: Scott Grady
  location: EMEA-IE
```

Tlgm: @it_boooks


```
  communities:
    - community: public123
      type: ro
    - community: supersecure
      type: rw
cisco@cisco:~/testing$
```


В текущем разделе мы не будем терять время, описывая все подробности и типы данных каждой YAML-переменной и обработку их средствами языка Jinja, так как все эти темы подробно рассматривались в главах 5 и 6.

Также выбраны определения информации о соединении и регистрационные данные, которые будут использоваться в следующих примерах для комплектов сценариев в файлах переменных уровня группы `all`.

```
ntc@ntc:~/testing$ more group_vars/all.yml
---

base_provider:
  username: ntc
  password: ntc123
  host: "{{ inventory_hostname }}"
ntc@ntc:~/testing$
```

Создание шаблонов Jinja В предыдущем разделе были определены данные SNMP, которые будут использоваться как входные данные при создании конфигурации. Теперь необходимо создать соответствующие шаблоны Jinja. Ansible автоматически выполняет поиск шаблонов в каталоге, из которого запускаются комплекты сценариев, а также в каталоге *templates*, расположенном в той же ветви файловой системы. Шаблоны SNMP будут храниться в подкаталоге *./templates/snmp*.

Можно представить предполагаемую схему хранения создаваемых файлов следующим образом:

```
ntc@ntc:~/testing$ tree templates/
templates/
├── snmp
    ├── eos.j2
    ├── ios.j2
    ├── junos.j2
    ├── nxos.j2
```

Ниже показан шаблон, созданный и сохраненный в файле *ios.j2*:

```
snmp-server location {{ snmp.location }}
snmp-server contact {{ snmp.contact }}
{% for community in snmp.communities %}
snmp-server community {{ community.community }} {{ community.type | upper }}
{% endfor %}
```

Содержимое шаблона IOS вполне понятно и не вызывает никаких вопросов. Здесь применен Jinja-фильтр `upper`, преобразующий в буквы верхнего регистра

Tlgm: @it_boooks

имена типов, ранее определенных как ro или rw в YAML-файле данных. При работе в командной строке IOS выполняется преобразование имен типов в RO и RW при вводе данных в конфигурацию. Важность этого преобразования станет более понятной после развертывания созданных конфигураций.

Ниже приведен шаблон Juniper для конфигурирования тех же данных SNMP на устройстве Junos:

```
set snmp location {{ snmp.location }}
set snmp contact {{ snmp.contact | replace(' ', '_') }}
{% for community in snmp.communities %}
{% if community.type | lower == "rw" %}
set snmp community {{ community.community }} authorization read-write
{% elif community.type | lower == "ro" %}
set snmp community {{ community.community }} authorization read-only
{% endif %}
{% endfor %}
```

Для этого шаблона Juniper необходимо отметить некоторые особенности:

- фильтр replace использован, поскольку Junos не поддерживает пространства (spaces) в своих контактах. Другой вариант – изменение самих данных, но в данном случае важно показать шаблон для одной ОС в сравнении с шаблоном для другой ОС;
- добавлено условное выражение, так как ro и rw не используются в командах Junos, поэтому необходимо установить соответствие исходных данных и корректных команд ОС Junos.

Далее показаны шаблоны для ОС EOS и NXOS:

```
snmp-server location {{ snmp.location }}
snmp-server contact {{ snmp.contact }}
{% for community in snmp.communities %}
snmp-server community {{ community.community }} {{ community.type }}
{% endfor %}
```

```
snmp-server location {{ snmp.location }}
snmp-server contact {{ snmp.contact }}
{% for community in snmp.communities %}
{% if community.type | lower == "rw" %}
snmp-server community {{ community.community }} group network-admin
{% elif community.type | lower == "ro" %}
snmp-server community {{ community.community }} group network-operator
{% endif %}
{% endfor %}
```

Генерация файлов конфигурации сети Файлы переменных с требуемыми значениями данных и файлы шаблонов готовы к работе, и можно перейти к завершающему этапу перед развертыванием – к генерации файлов конфигурации SNMP. Для этого воспользуемся модулем Ansible с именем template. Этот модуль автоматизирует создание файлов, обрабатывая входные данные (переменные) с помощью шаблонов Jinja.

В следующем примере показано, как применяется модуль template. Для этого модуля используются параметры src и dest. Параметр src определяет нужный

Tlgm: @it_boooks

шаблон, который будет использоваться для обработки, параметр dest указывает локацию, в которой будут сохранены полностью обработанные версии файлов конфигурации.

```
---

  - name: PLAY 1 - GENERATE SNMP CONFIGURATIONS
    hosts: all
    connection: local
    gather_facts: no

    tasks:

      - name: GENERATE CONFIGS FOR EACH OS
        template:
          src: "./snmp/{{ os }}.j2"
          dest: "./configs/snmp/{{ inventory_hostname }}.cfg"
```



В нашей среде для тестирования мы вручную создали каталог configs и подкаталог snmp в нем. Эту процедуру также можно автоматизировать с помощью Ansible, воспользовавшись модулем file.

В приведенном выше примере отметим, что созданные переменные также можно использовать при указании путей, при извлечении значений или для каких-либо других целей в комплекте сценариев. Благодаря этому,в любой отдельной задаче комплекта сценариев можно автоматически сгенерировать требуемые конфигурации для любого количества устройств вне зависимости от типа ОС.



Напомним, что ранее в текущем разделе при формировании inventory-файла была добавлена переменная os как переменная уровня группы.

Теперь комплект сценариев полностью готов к выполнению, то есть к генерации требуемых файлов конфигурации. Продемонстрируем процедуру генерации на следующем примере:

```
cisco@cisco:~/testing$ ansible-playbook -i inventory snmp.yml
```

После выполнения комплекта сценариев автоматически создаются следующие файлы, полученные в результате подстановки соответствующих значений переменных в подготовленные шаблоны Jinja.

```
ntc@ntc:~/testing$ tree configs/snmp/
configs/snmp/
├── csr1.cfg
├── csr2.cfg
├── eos-spine1.cfg
├── eos-spine2.cfg
├── nxos-spine1.cfg
├── nxos-spine2.cfg
├── vmx1.cfg
└── vmx2.cfg

0 directories, 8 files
```

Tlgm: @it_boooks

Для проверки правильности подстановки значений переменных из каждого YAML-файла выведем результаты обработки для одного устройства из региона AMERS и для одного устройства из региона EMEA:

```
ntc@ntc:~/testing$ cat configs/snmp/csr1.cfg
snmp-server location AMERICAS-NJ
snmp-server contact Joe Smith
snmp-server community public RO
snmp-server community public123 RO
snmp-server community private RW
snmp-server community secure RW
ntc@ntc:~/testing$

ntc@ntc:~/testing$ cat configs/snmp/vmx1.cfg
set snmp location EMEA-IE
set snmp contact Scott_Grady
set snmp community public123 authorization read-only
set snmp community supersecure authorization read-write
ntc@ntc:~/testing$
```

Создание шаблонов, переменных и комплекта сценариев, состоящего из единственной задачи, является одним из самых распространенных способов использования Ansible на начальном этапе освоения, поскольку при этом фактически не требуются реальные сетевые устройства. Такая методика позволяет постепенно улучшать навыки работы с языками Jinja и YAML в процессе создания и стандартизации конфигураций сетевых устройств.

Проверка существования конфигурации

В предыдущем примере были разработаны и автоматически сгенерированы восемь файлов конфигурации. В следующем примере демонстрируется развертывание этих конфигураций и проверка их существования на каждом устройстве.

Концепция идемпотентности В процессе работы над нашим учебным примером мы с особым вниманием относимся к использованию слов и фраз надлежащим образом, например фраза «конфигурирование SNMP» отличается от фразы «проверка существования конфигурации SNMP». В обычном скрипте на языке Python можно передавать SNMP-команды при каждом выполнении этого скрипта. В Ansible, как и во многих инструментальных средствах управления конфигурациями DevOps, методика управления конфигурациями должна обеспечивать *идемпотентность* (*idempotency*), то есть свойство неизменной повторяемости одного и того же результата при многократном применении к объекту одной и той же операции, а изменения вносятся, только если это действительно необходимо. В рассматриваемом здесь контексте сетевой среды каждый модуль усиливает «разумность» методики, позволяющей выполнять проверку того, что конфигурационные команды передаются на устройство, только если они действительно необходимы для приведения этого устройства в требуемое состояние.

Tlgm: @it_boooks

На высшем уровне модули выполняют эту задачу посредством предварительного сбора информации о конфигурации, существующей на устройстве. Команды, которые по вашему мнению непременно должны существовать на устройстве, сравниваются с текущим состоянием (работающая конфигурация). Только в том случае, когда требуемые команды отсутствуют в текущей конфигурации, они передаются на устройство. Такой подход более безопасен в том плане, что при многократном выполнении комплекта сценариев модуль передает команды только один раз.

Именно так работают модули типа `config` в Ansible. По умолчанию они получают информацию с помощью команды `show run`, а команды из комплекта сценариев передаются на устройство только в том случае, когда они отсутствуют в выводе команды `show run`.

Использование модуля типа config Рассмотрим практическое применение модуля `eos_config` для развертывания файла конфигурации SNMP для устройств Arista EOS.

```
- name: PLAY 2 - ENSURE EOS SNMP CONFIGS ARE DEPLOYED
  hosts: eos
  connection: local
  gather_facts: no

  tasks:

    - name: DEPLOY CONFIGS FOR EOS
      eos_config:
        src: "./configs/snmp/{{ inventory_hostname }}.cfg"
        provider: "{{ base_provider }}"
```

Приведенный выше сценарий был добавлен в существующий комплект сценариев, генерирующий конфигурации, как было показано выше в текущем разделе. Для этого сценария определено имя `PLAY 2`. Кроме того, можно было бы создать новый комплект сценариев, предназначенный исключительно для развертывания конфигураций.

Модули типа `config` принимают множество разнообразных параметров, но приведенная в примере задача использует лишь два параметра из этого обширного набора: `provider` и `src`. Здесь используется ссылка на переменную `base_provider`, определенную в подгруппе `group_vars` для группы `all`, а значение этой переменной передается в модуль как значение параметра `provider`. Это упрощает формирование всех задач в комплекте сценариев, так как исключает необходимость определения вручную словаря `provider` N раз, если в комплекте сценариев содержится N задач.

Параметр `src` может ссылаться на файл конфигурации или непосредственно на шаблон Jinja. В приведенном выше примере `src` явно ссылается на файл, сгенерированный в сценарии из самого первого примера текущего раздела.



В модулях типа `config` используются и другие параметры, среди которых можно выделить следующие:

Tlgm: @it_boooks

- `commands` – невозможно использовать этот параметр вместе с `src` (взаимоисключающие параметры). Вместо ссылки на файл шаблона или конфигурации позволяет определить список команд непосредственно в комплекте сценариев;
- `parents` – список «родительских» команд, определяющих иерархию, в которой должны быть выполнены основные команды (требуется при использовании параметра `commands` и конфигурировании устройства с помощью команд, не входящих в набор команд глобального режима конфигурации). Пример родительской команды: `['interface Eth1']`; пример списка соответствующих команд: `['duplex full']` – если выполняется конфигурация полнодуплексного режима для интерфейса Eth1.

Для модулей типа `config` существует множество других параметров, например параметры, позволяющих выполнять специальные команды до и после выполнения команд, определенных в параметрах `parents` и `commands`. Напомним, что для получения подробной информации обо всех поддерживаемых параметрах с практическими примерами их применения необходимо воспользоваться командой `ansible-doc <os>_config`.

Режимы проверки, подробного вывода и ограниченный режим Перед выполнением комплекта сценариев, создающего и развертывающего конфигурации, рассмотрим несколько функциональных режимов, которые необходимо знать при работе с комплектами сценариев Ansible:

- *режим проверки (check mode)* – это возможность запускать комплекты сценариев в режиме испытания или тестирования (dry run), то есть возможность узнать, *будут ли* произведены какие-либо изменения. При этом совершаются все ожидаемые действия при выполнении проверяемой задачи, но в действительности связанные с этим изменения не происходят. Для использования режима проверки в команду выполнения комплекта сценариев добавляется флаг `--check`. Примечание: режим проверки является функцией для отдельных модулей;
- *режим подробного вывода (verbosity)* – каждый модуль возвращает данные в формате JSON, которые содержат метаданные о текущей выполняемой задаче. Для модулей типа `config` в этих данных содержатся команды, которые необходимо передать на устройство. Модули типа `command` возвращают ответ устройства. Для запуска комплекта сценариев в режиме подробного вывода, чтобы увидеть JSON-данные, возвращаемые каждым модулем, в команду выполнения комплекта сценариев добавляется флаг `-v`. При поиске и устранении возникших проблем в флаге можно использовать до четырех символов v (`-vvvv`), определяющих уровень подробности вывода;
- *ограниченный режим (limit)* – обобщенный вариант автоматизации внесения изменений в устройства – использование в сценарии из комплекта определений хостов `hosts`, например `hosts: all`. Если необходима автоматизация только устройств `junos`, то один из вариантов вносимых изменений – `hosts: junos`. Разумеется, это зависит от наличия определения `junos` в inventory-файле. Другой вариант – использование флага `--limit` при выполнении комплекта сценариев, например `--limit junos`. Примечание: значение, передаваемое вместе с флагом `--limit`, обязательно должно со-

Tlgm: @it_boooks

держаться в группе или в группах, уже определенных в ключе hosts в соответствующем комплекте сценариев. Следует отметить, что можно также передавать имя одного устройства, группы или нескольких устройств или групп, например: `--limit junos,eos,csr1`.

В следующем примере показано, как можно использовать все три описанных выше флага при выполнении одного комплекта сценариев. Сочетание режима проверки с режимом подробного вывода особенно полезно при развертывании конфигураций, поскольку показывает в точности те команды, которые *будут* переданы на устройство, но в действительности развертывание не выполняется.

```
ntc@ntc:~/testing$ ansible-playbook -i inventory snmp.yml --limit eos-spine1 --check -v
Using /etc/ansible/ansible.cfg as config file

PLAY [PLAY 1 - GENERATE SNMP CONFIGURATIONS] ********************************

TASK [GENERATE CONFIGS FOR EACH OS] *****************************************
ok: [eos-spine1] => {"changed": false, "gid": 1000, "group": "cisco",
"mode": "0664", "owner": "cisco", "path": "./configs/snmp/eos-spine1.cfg",
"size": 133, "state": "file", "uid": 1000}

PLAY [PLAY 2 - ENSURE EOS SNMP CONFIGS ARE DEPLOYED] ************************

TASK [DEPLOY CONFIGS FOR EOS] ***********************************************

changed: [eos-spine1] => {"changed": true, "commands": ["snmp-server location EMEA-IE",
"snmp-server contact Scott Grady", "snmp-server community public123 ro",
"snmp-server community supersecure rw"], "session": "ansible_1497404657",
"updates": ["snmp-server location EMEA-IE", "snmp-server contact Scott Grady",
"snmp-server community public123 ro", "snmp-server community supersecure rw"]}

PLAY RECAP ******************************************************************
eos-spine1                 : ok=2    changed=1    unreachable=0    failed=0

ntc@ntc:~/testing$
```

В этом примере можно видеть более подробный вывод результатов, так как комплект сценариев был выполнен в соответствующем режиме, что позволяет увидеть данные в формате JSON, возвращаемые каждой задачей и каждым модулем. При изучении вывода выполненого комплекта сценариев очень важно понимать, на какие данные указывают параметры ok и changed. В последней строке предыдущего примера указаны данные `ok=2 changed=1`. Это означает, что были успешно выполнены (ok) две задачи, но только в одной из них должны будут произойти изменения (в действительности изменений не было, так как задача выполнялась в режиме проверки). При повторном выполнении полностью идемпотентного комплекта сценариев и при всех последующих выполнениях результат всегда будет одинаковым – `changed=0`, поскольку никаких изменений происходить не должно.

Теперь можно без затруднений добавить все последующие сценарии, которые будут развертывать конфигурации SNMP для каждой ОС. Ниже приведен пример сценария для Junos:

Tlgm: @it_boooks


```
- name: PLAY 3 - ENSURE JUNOS SNMP CONFIGS ARE DEPLOYED USING SET COMMANDS
  hosts: junos
  connection: local
  gather_facts: no

  tasks:

    - name: DEPLOY CONFIGS FOR JUNOS
      junos_config:
        src: "./configs/snmp/{{ inventory_hostname }}.cfg"
        provider: "{{ base_provider }}"
```

Здесь можно видеть, что в этом сценарии единственным отличием является имя используемого модуля. Никаких других отличий от предыдущего сценария нет.

Сбор и просмотр сетевых данных

Ansible, как и многие другие инструментальные средства автоматизации, часто используется для развертывания конфигураций. Но при работе с Ansible также предоставляется возможность автоматизации сбора данных, получаемых от сетевых устройств. Мы подробно рассмотрим два главных метода сбора данных: использование модулей ядра `facts` и выполнение разнообразных команд `show` с помощью модуля `command`.

Использование модулей ядра facts Сначала рассмотрим работу с модулями `facts`. Модули ядра `facts` возвращают данные, описанные в табл. 9.1.

Таблица 9.1. Модули ядра facts, используемые для автоматизации сбора данных, получаемых от сетевых устройств

Модуль ядра типа facts	Получаемый результат
`ansible_net_model`	Имя модели, возвращаемое с устройства
`ansible_net_serialnum`	Серийный номер удаленного устройства
`ansible_net_version`	Версия операционной системы, работающей на удаленном устройстве
`ansible_net_hostname`	Сконфигурированное имя хоста для данного устройства
`ansible_net_image`	Имя image-файла для работающего устройства
`ansible_net_filesystems`	Имена всех файловых систем, доступных на устройстве
`ansible_net_memfree_mb`	Объем доступной свободной памяти на удаленном устройстве, Мб
`ansible_net_memtotal_mb`	Общий объем памяти на удаленном устройстве, Мб
`ansible_net_config`	Текущая активная конфигурация устройства
`ansible_net_all_ipv4_addresses`	Все IPv4-адреса, сконфигурированные для устройства
`ansible_net_all_ipv6_addresses`	Все IPv6-адреса, сконфигурированные для устройства
`ansible_net_interfaces`	Хеш-значения всех интерфейсов, работающих в системе
`ansible_net_neighbors`	Список соседних устройств по протоколу LLDP, получаемый от удаленного устройства

После запуска задачи, предназначенной для сбора информации (фактов), можно получить доступ к каждому из перечисленных выше ключей непосред-

Tlgm: @it_boooks

ственно в комплекте сценариев или в шаблоне Jinja точно так же, как и к любой другой переменной (обычно с использованием двойных фигурных скобок).

Рассмотрим пример.

```
---

  - name: PLAY 1 - COLLECT FACTS FOR IOS
    hosts: iosxe
    connection: local
    gather_facts: no

    tasks:

      - name: COLLECT FACTS FOR IOS
        ios_facts:
          provider: "{{ base_provider }}"
```

Вы можете самостоятельно добавить другие сценарии для сбора информации с каждого устройства с учетом конкретной операционной системы.

Использование модуля отладки debug Для просмотра фактов (информации), возвращаемых из модуля, можно запустить комплект сценариев в режиме подробного вывода или воспользоваться модулем отладки `debug` с параметром `var`, указывающим на корректный ключ `facts`, как показано ниже:

```
# определение сценария пропущено
    tasks:

      - name: COLLECT FACTS FOR IOS
        ios_facts:
          provider: "{{ base_provider }}"

      - name: DEBUG OS VERSION
        debug:
          var: ansible_net_version

      - name: DEBUG HOSTNAME
        debug:
          var: ansible_net_hostname
```



Обычно обращение к переменным выполняется с помощью двойных фигурных скобок как в комплекте сценариев, так и в шаблоне Jinja. Использование модуля `debug` с параметром `var` представляет собой один из тех случаев, когда синтаксис с двойными фигурными скобками не применяется. Следует также помнить, что любая переменная в комплекте сценариев также доступна и в шаблоне Jinja.

Запуск задач из предыдущего примера дает следующие результаты:

```
TASK [COLLECT FACTS FOR IOS] ***************************************************
ok: [csr1]

TASK [DEBUG OS VERSION] ********************************************************
ok: [csr1] => {
    "ansible_net_version": "16.3.1",
    "changed": false
```

Tlgm: @it_boooks


```
}

TASK [DEBUG HOSTNAME] ***********************************************************
ok: [csr1] => {
    "ansible_net_hostname": "csr1",
    "changed": false
}
```

Здесь можно видеть, что выводятся значения именно тех переменных, которые были явно указаны в командах `debug`.

Выполнение команд show и запись данных в файл

Теперь вы знаете, как собирать информацию (факты) и выполнять отладку возвращаемых данных. В этом разделе рассматривается та же операция, реализованная с помощью модуля ядра `command`. Демонстрируется выполнение команд `show`, отладка ответа с последующей записью вывода команд `show` в файл.

Напомним, что одним из способов просмотра возвращаемых данных в формате JSON в любой задаче является простое выполнение комплекта сценариев в режиме подробного вывода. Существует и другой способ, позволяющий воспользоваться модулем `debug`, – при таком подходе обязательно необходимо сначала сохранить полученный от рабочего модуля ответ в формате JSON в переменной, затем начать отладку этой переменной. Примечание: нет необходимости сохранять факты (facts) в отдельной переменной, так как факты сами по себе доступны для непосредственного использования в среде Ansible.

Использование атрибута задачи register Чтобы сохранить данные в формате JSON, возвращаемые модулем, можно воспользоваться атрибутом задачи `register`. Это позволит сохранить полученные JSON-данные как переменную типа словарь (dictonary).

Атрибут задачи вставляется в сценарий на том же уровне форматирования (отступа), что и имя модуля. В приведенном ниже примере используется модуль `ios_command`, следовательно, атрибут `register` записывается как ключ на уровне форматирования, соответствующего `ios_command`. Связанное с атрибутом register значение представляет собой имя переменной, в которой предполагается сохранить возвращаемые данные.

```
- name: ISSUE SHOW COMMAND
  ios_command:
    commands:
      - show run | inc snmp-server community
    provider: "{{ base_provider }}"
  register: snmp_data
```

После выполнения комплекта сценариев значением переменной `snmp_data` является JSON-объект, возвращенный модулем `ios_command`, который также можно увидеть при запуске комплекта сценариев в режиме подробного вывода.

После того как переменная `snmp_data` создана или *зарегистрирована* (*registered*), можно использовать модуль `debug` для просмотра сохраненных в ней

Tlgm: @it_boooks

данных. В следующем примере показан результат, выводимый при выполнении комплекта сценариев, только для задачи из предыдущего примера.

```
TASK [DEBUG COMPLETE SNMP RESPONSE] ****************************************
ok: [csr1] => {
    "changed": false,
    "snmp_data": {
        "changed": false,
        "stdout": [
            "snmp-server community ntc RO\nsnmp-server
            community public RO\nsnmp-server community public123 RO\nsnmp-server
            community private RW\nsnmp-server community secure RW"
        ],
        "stdout_lines": [
            [
                "snmp-server community ntc RO",
                "snmp-server community public RO",
                "snmp-server community public123 RO",
                "snmp-server community private RW",
                "snmp-server community secure RW"
            ]
        ]
    }
}
```

Здесь можно видеть, что использование атрибута `register` совместно с модулем `debug` предоставляет удобный способ сохранения и просмотра возвращаемых данных. При просмотре необходимо понять структуру данных, чтобы точно знать, как организовать доступ к требуемым данным. Например, модули типа `command` возвращают ключи `stdout` и `stdout_lines`. Значением каждого ключа является список: для `stdout` это список ответов на команды с указанием длины списка, равной количеству команд, переданных на устройство. Каждый элемент списка – это ответ на соответствующую выполненную команду. Значение ключа `stdout_lines` представляет собой объект более сложной структуры (с вложенными объектами), в котором внешний объект (самого высокого уровня) – это список, а внутренние объекты могут быть различными в зависимости от используемого транспортного протокола: список для CLI/SSH или словарь для API (например, для NX-API или eAPI). Мы сосредоточимся на использовании ключа `stdout`.

Если необходимо отлаживать только реальный ответ как строку, то следует использовать одну из приведенных ниже команд отладки:

```
- name: DEBUG COMMAND STRING RESPONSE WITH JINJA SHORTHAND SYNTAX
  debug:
    var: snmp_data.stdout.0

- name: DEBUG COMMAND STRING RESPONSE WITH STANDARD PYTHON SYNTAX
  debug:
    var: snmp_data['stdout'][0]
```

Tlgm: @it_boooks

Кроме того, как уже было отмечено выше, каждая переменная в комплекте сценариев также доступна и в шаблоне, поэтому если нужно записать данные в файл, то можно воспользоваться модулем `template` с простейшим типовым шаблоном, показанным ниже:

```
{{ snmp_data['stdout'][0] }}
```

После этого возможно использование единственной задачи для записи данных в файл с помощью приведенного выше шаблона:

```
- name: WRITE DATA TO FILE
  template:
    src: basic.j2
    # этот шаблон был сохранен в каталоге templates
    dest: ./commands/snmp/{{ inventory_hostname }}.txt
    # каталоги commands и snmp были созданы вручную
```

Чрезвычайно важно понимать, что каждый модуль возвращает данные в формате JSON и эти данные сохраняются с помощью атрибута задачи `register`. Описанная методика применяется не только для отладки данных или записи данных в файл, ее можно также использовать для выполнения проверок согласованности и генерации отчетов. Эти темы рассматриваются в следующих разделах.

Выполнение проверок согласованности

Проверки согласованности (compliance checks) достаточно часто выполняются вручную посредством установления SSH-соединения с устройством и верификации параметров и характеристик (разрешены или запрещены, сконфигурированы или не сконфигурированы), чтобы узнать, выполнены ли требования к конкретной сетевой среде и требования по обеспечению ее безопасности. Автоматизация этих типов проверок оптимизирует процесс проверки, позволяющий убедиться в том, что конфигурация и рабочее состояние всегда соответствуют требованиям и ожиданиям. Такие процедуры также весьма полезны для инженеров по обеспечению безопасности, например при проверке полного соответствия требований к надежности и защищенности устройств.

При подготовке к выполнению проверок согласованности с помощью Ansible сначала необходимо рассмотреть еще две концепции, описанные ниже:

- `set_fact` – это модуль, создающий динамические переменные (для временного употребления) из некоторого другого набора данных сложной структуры. Например, если уже зарегистрирована новая переменная, представляющая собой большой словарь, то можно выделить из этого объекта только одну интересующую нас пару ключ-значение. Использование модуля `set_fact` позволяет сохранять одно из многочисленных значений сложного объекта как новый факт или новую переменную;
- `assert` – в разработке программного обеспечения широко применяются команды `assert` для проверки результата определенного условного выражения – `True` или `False`. В Ansible можно воспользоваться модулем `assert`,

Tlgm: @it_boooks

чтобы проверить, является ли значением некоторого условного выражения `True` или `False`.

Рассмотрим пример проверки (asserting) того факта, что виртуальная сеть VLAN 20 сконфигурирована на двух коммутаторах Arista EOS. Пример состоит из следующих, последовательно выполняемых задач:

1. Сбор данных о VLAN.
2. Сохранение полученных данных о VLAN в переменной `vlan_data`.
3. Вывод (отладка) всех данных о VLAN, чтобы просмотреть полученное содержимое.
4. Извлечение только идентификаторов VLAN ID из полного набора полученных данных.
5. Вывод только идентификаторов VLAN ID (проверка того факта, что запрошенная подсистема работает в полном соответствии с требованиями и ожиданиями).
6. Выполнение проверки (assertion) того факта, что VLAN 20 действительно содержится в списке виртуальных сетей VLAN.

Ниже приведен комплект сценариев, выполняющий все перечисленные выше задачи:

```
---

  - name: PLAY 1 - ISSUE SHOW COMMANDS
    hosts: eos
    connection: local
    gather_facts: no

    tasks:

      - name: RETRIEVE VLANS JSON RESPONSE
        eos_command:
          commands:
            - show vlan brief | json
          provider: "{{ base_provider }}"
        register: vlan_data

      - name: DEBUG VLANS AS JSON
        debug:
          var: vlan_data

      - name: CREATE EXISTING_VLANS FACT TO SIMPLIFY ACCESSING VLANS
        set_fact:
          existing_vlan_ids: "{{ vlan_data.stdout.0.vlans.keys() }}"

      - name: DEBUG EXISTING VLAN IDs
        debug:
          var: existing_vlan_ids

      - name: PERFORM COMPLIANCE CHECKS
        assert:
        that:
          - "'20' in existing_vlan_ids"
```

Tlgm: @it_boooks

При выполнении этот комплект сценариев выводит следующий результат (показан вывод результата для самой последней задачи), показывающий, что виртуальная сеть VLAN сконфигурирована только на коммутаторе eos-spine1, но не на eos-spine2.

```
TASK [PERFORM COMPLIANCE CHECKS] ***********************************************
fatal: [eos-spine2]: FAILED! => {
    "assertion": "'20' in existing_vlan_ids",
    "changed": false,
    "evaluated_to": false,
    "failed": true
}
ok: [eos-spine1] => {
    "changed": false,
    "msg": "All assertions passed"
}
```

При полном понимании того, какие данные должны возвращаться из конкретной задачи и команд типа `show`, можно выполнять бесконечное количество проверок всех необходимых условий и требований.

В следующем разделе демонстрируется возможность автоматической генерации отчетов на основе данных, возвращаемых сетевыми устройствами.

Генерация отчетов

В этом разделе продолжается развитие темы сбора данных. Выше были показаны возможности сбора фактов (facts) данных, использование команд типа `show`, регистрация данных, запись данных в файл и выполнение оперативных проверок (assertions). В текущем разделе мы вернемся к процедуре записи данных в файл, но с новой целью – генерация отчета.

В предыдущем разделе демонстрировалась процедура сбора фактов с использованием модулей ядра `facts`. В одном комплекте были созданы три полноценных сценария, которые собирали факты с помощью модулей `ios_facts`, `eos_facts` и `nxos_facts`. В следующем примере в комплект добавлен четвертый сценарий, выполняющий автоматическую генерацию отчета по собранным фактам.



В приведенном ниже примере отчет составлен только по фактам, полученным от указанных устройств, но точно такая же методика может быть применена для любых данных, возвращаемых командами `show`, или полученных из любой другой переменной, существующей в проекте Ansible.

```
- name: PLAY 4 - CREATE REPORTS
  hosts: "iosxe,eos,nxos"
  connection: local
  gather_facts: no

  tasks:

    - name: GENERATE DEVICE SPECIFIC REPORTS
      template:
```

Tlgm: @it_boooks


```
        src: ./reports/facts.j2
        # these sub-directories were created manually
        dest: ./reports/facts/{{ inventory_hostname }}.md

    - name: CREATE MASTER REPORT
      assemble:
        src: ./reports/facts/
        dest: ./reports/master-report.md
        delimiter: "---"
      run_once: true
```

В этом комплекте сценария некоторые детали требуют более подробного объяснения:

- это пример автоматизации трех групп устройств, определенных в inventory-файле как `hosts: "iosxe,eos,nxos"`;
- шаблон задачи генерирует отчет на языке разметки Markdown (*.md*) для каждого устройства;
- модуль `assemble` объединяет все отдельные отчеты в единый обобщенный (главный) отчет. В этой задаче представлен еще один атрибут `run_once`. С технической точки в нем нет необходимости, но, как вы помните, в этом сценарии выполняется автоматизация нескольких хостов, а необходим только один главный сводный отчет. Поэтому системе Ansible просто передается инструкция по запуску процедуры для первого автоматизируемого устройства, а поскольку модуль обладает свойством идемпотентности, то даже при выполнении его N раз без атрибута `run_once` на систему не должно оказываться никакого воздействия. Для полной уверенности можно также добавить атрибут задачи `delegate_to`, то есть использовать строку `delegate_to: localhost`, чтобы выполнить модуль для системы (localhost) вместо однократного запуска на первом автоматизируемом хосте. В данном случае это имеет смысл, поскольку обе опции работают и решают проблему;
- группа из трех дефисов `---` в языке разметки Markdown обозначает горизонтальную линию на всю ширину страницы. Этот элемент разметки следует использовать как разделитель между отчетами по отдельным устройствам, объединеным в сводном отчете.

Ниже приведен шаблон, используемый для генерации отчета о собранных фактах:

```
# {{ inventory_hostname }}

## Facts

Serial Number: {{ ansible_net_serialnum }}

OS Version:   {{ ansible_net_version }}

## Neighbors

| Device | Local Interface | Neighbor | Neighbor Interface |
|--------|-----------------|----------|--------------------|
```

Tlgm: @it_boooks


```
{% for interface, neighbors in ansible_net_neighbors.items() %}
{% for neighbor in neighbors %}
| {{ inventory_hostname }} | {{ interface }} | {{ neighbor.host }} | {{ neighbor.port }} |
{% endfor %}
{% endfor %}

## Interface List
{% for interface in ansible_net_interfaces.keys() %}
  - {{ interface }}
{% endfor %}
```

С помощью приведенных в примере синтаксических конструкций показано, как можно сформировать таблицу средствами языка разметки Markdown. Она обрабатывается как HTML-таблица, если передать ее на GitHub или воспользоваться просмотрщиком, поддерживающим разметку Markdown. В большинстве веб-браузеров имеются доплнительно подключаемые модули для просмотра файлов Markdown. Кроме того, можно воспользоваться текстовым редактором с поддержкой формата Markdown, например StackEdit (https:// stackedit.io/editor – версия 4; https://stackedit.io/app# – версия 5).

После обработки вывод текстового отчета для одного устройства будет выглядеть приблизительно следующим образом:

```
# csr1

## Facts

Serial Number: 9KXI0D7TVFI

OS Version:   16.3.1

## Neighbors

| Device | Local Interface |      Neighbor      | Neighbor Interface |
|--------|-----------------|--------------------|--------------------|
| csr1   | Gi4             | csr2.ntc.com       | Gi4                |
| csr1   | Gi1             | csr2.ntc.com       | Gi1                |
| csr1   | Gi1             | eos-spine1.ntc.com | Management1        |
| csr1   | Gi1             | vmx1               | fxp0               |
| csr1   | Gi1             | eos-spine2.ntc.com | Management1        |
| csr1   | Gi1             | vmx2               | fxp0               |

## Interface List
  - GigabitEthernet4
  - GigabitEthernet1
  - GigabitEthernet2
  - GigabitEthernet3
```

Если передать обработанный файл с разметкой Markdown на GitHub, то можно посмотреть результат его обработки непосредственно на сервисе GitHub в режиме онлайн (см. рис. 9.2).

Tlgm: @it_boooks

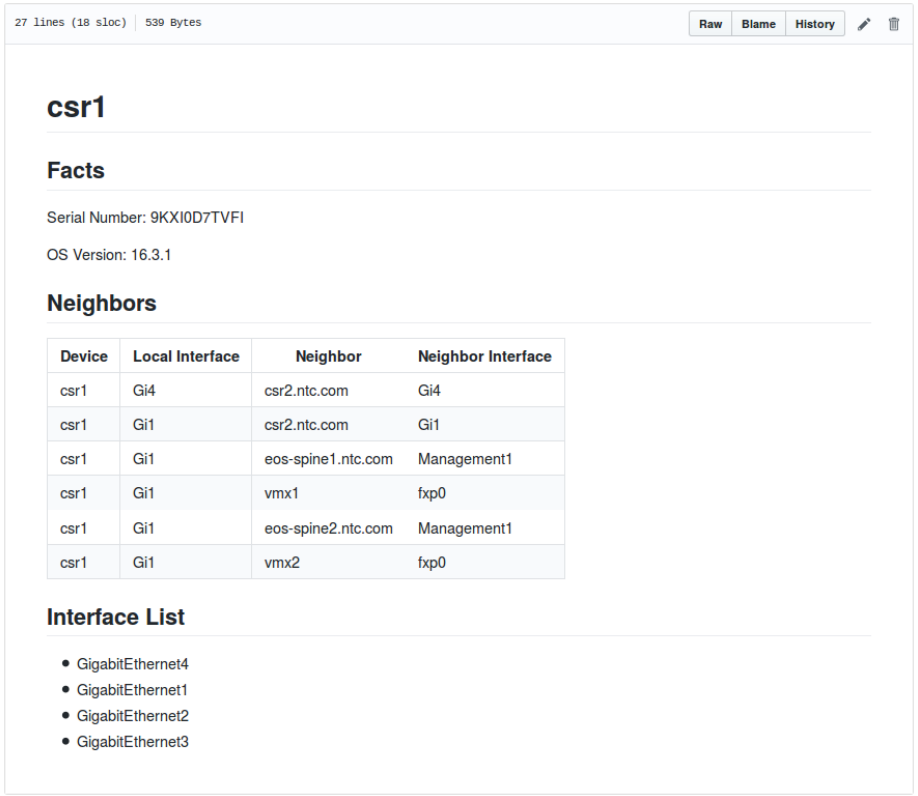


Рис. 9.2 ❖ Просмотр сгенерированного отчета о собранных фактах на сервисе GitHub

Можно создать любой требуемый тип шаблона: здесь рассматривались только шаблоны конфигурации и шаблоны Markdown, но вы можете самостоятельно создавать шаблоны HTML для еще более тонкой настройки отчета.

Использование сторонних модулей Ansible от независимых авторов

Во всех рассмотренных ранее в текущей главе примерах использовались модули ядра Ansible, то есть модули, с которыми можно работать сразу после установки системы Ansible. С помощью этих модулей можно выполнить огромный объем работы – от процедур конфигурации и проверок согласованности до генерации отчетов, как вы сами убедились. Тем не менее существует весьма активное сообщество независимых авторов модулей для Ansible, предназначенных для автоматизации сети. В этом разделе рассматриваются две основные группы сто-

Tlgm: @it_boooks

ронних модулей с открытым исходным кодом от независимых авторов, а также управление установкой таких модулей на основном хосте Ansible.



Многие модули с открытым исходным кодом дополняют функциональность, обеспечиваемую ядром Ansible. Следует также отметить, что многие из этих сторонних модулей появились раньше, чем сетевые модули ядра Ansible.

Модули NTC

Два года назад компания Network to Code (NTC) открыла исходный код некоторых модулей Ansible для работы с набором устройств от нескольких различных производителей. Теперь эти модули часто называют просто модули NTC (https://github.com/networktocode/ntc-ansible).

Комплект модулей NTC наиболее часто используется благодаря следующим трем основным функциональным характеристикам:

- автоматический синтаксический разбор (парсинг) необработанного текста, выводимого устаревшими (legacy) устройствами, с использованием предварительно созданных шаблонов TextFSM. TextFSM упрощает применение регулярных выражений в команде вывода данных. Исходный код всех шаблонов также открыт и опубликован на GitHub (https://github.com/networktocode/ntc-templates). Парсинг выполняется с помощью модуля `ntc_show_command`, который в действительности представляет собой обертку для связки netmiko + TextFSM. Для этого модуля также предусмотрен режим офлайн, который позволяет воспользоваться модулями типа `command`, записывать данные в файл с последующим синтаксическим разбором этих данных средствами того же модуля;
- выполнение на устройствах команд, которые пока еще не поддерживаются ядром Ansible. Поскольку внутри модулей `ntc_show_command` и `ntc_config_command` используется библиотека netmiko, можно автоматизировать любое устройство, поддерживающее netmiko (все операции выполняются через SSH-соединение). Библиотека netmiko поддерживает более двадцати типов устройств, поэтому функциональность вышеназванных модулей весьма надежна;
- обеспечение управления операционной системой устройства. В комплекте NTC имеется несколько модулей, позволяющих выполнять резервное копирование файлов конфигурации, сохранять текущие конфигурации, копировать файлы на устройства, обновлять образы ОС и перезагружать их. Эти модули достаточно часто используются для обновлений IOS и NXOS, но в них также обеспечивается определенный уровень поддержки EOS и Junos.

Модули NAPALM

Проект Network Automation and Programmability Abstraction Layer with Multivendor Support (NAPALM) (https://napalm-automation.net) представляет собой постоянно развивающееся сообщество, ведущее разработку комплексных решений с открытым исходным кодом для процесса сетевой автоматизации

Tlgm: @it_boooks

с поддержкой одновременного использования устройств от различных производителей сетевого оборудования.

Более подробно NAPALM рассматривается в приложении Б данной книги, но здесь следует отметить, что в последние два года активному использованию модулей NAPALM (https://github.com/napalm-automation/napalm-ansible) для Ansible способствуют две причины:

- *декларативная концепция управления конфигурацией* (*declarative conigura-tion management*) – главное внимание система NAPALM уделяет требуемому состоянию конфигурации, предлагая для этого обобщенный механизм с использованием в основном «разумности», присущей самим сетевым устройствам, для реализации требуемого состояния файла конфигурации. При управлении абсолютно всеми файлами конфигурации по такой методике вы просто развертываете новую конфигурацию (ту, которую намерены применить). При этом не отправляются запрещающие команды («no»-команды). NAPALM абстрагируется от уровня выполнения конкретных команд на устройстве конкретного производителя и организует всю работу так, что нет необходимости заниматься микроменеджментом конфигураций отдельных устройств. Кроме того, имеется возможность передачи частичных конфигураций на основе поддержки сетевой ОС. И в этом случае выполняются только те команды, которые до этого не существовали на данном устройстве. Все перечисленные выше операции выполняются с помощью модуля `napalm_install_config`;
- *получение конфигурации и текущего рабочего состояния от устройств* – в NAPALM также имеется модуль `napalm_get_facts`, используемый для получения основного набора фактов и некоторой другой информации, например записи таблицы маршрутизации, таблица MAC-адресов, BGP-соседи, LLDP-соседи и многое другое. Преимущество использования этого модуля состоит в том, что выполняются предварительный парсинг и нормализация данных, получаемых от всех поддерживаемых типов устройств от различных производителей, тем самым исключается необходимость внесения каких-либо изменений в ваш комплект сценариев.



И модули NAPALM, и модули NTC обеспечивают работу с разнообразными типами устройств от различных производителей сетевого оборудования. Для модулей существует специальный параметр, определяющий операционную систему автоматизируемого устройства. Здесь нет модулей, предназначенных для каждой конкретной ОС, в отличие от ядра Ansible.

Как узнать о наличии сторонних модулей, не входящих в состав ядра

Если модуль поддержки устройства от какого-либо производителя сетевого оборудования не найден в ядре Ansible, то рекомендуется проверить страницу этого производителя на сервисе GitHub. В настоящее время существует несколько компаний, которые пока еще не передали свои модули в ядро Ansible:

Tlgm: @it_boooks

- HPE для коммутаторов Comware7;
- Juniper для устройств Junos;
- Citrix для устройств Netscaler;
- Palo Alto для комплектов PAN Security.

Поэтому при поиске модулей для управления конкретной платформой от какого-либо производителя оборудования рекомендуется всегда начинать с сервиса GitHub.

Установка сторонних модулей

Установка сторонних модулей с открытым исходным кодом или специализированных модулей выполняется достаточно просто. Для этого нужно выполнить следующие действия:

1. Выбрать путь в файловой системе Linux, то есть локацию для хранения всех сторонних модулей.
2. Перейти в эту локацию (по выбранному пути) и выполнить команду `git clone` в репозитории каждого модуля, который предполагается использовать.
3. Открыть файл конфигурации Ansible (*ansible.cfg*) и добавить в путь поиска модулей новый каталог, в котором выполнялось клонирование.
4. Если место расположения файла *ansible.cfg* неизвестно, то необходимо выполнить команду `ansible --version` в своей системе. Вывод будет выглядеть приблизительно так:

```
ntc@ntc:~/testing$ ansible --version
ansible 2.3.0.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/etc/ntc/ansible/']
  python version = 2.7.6 (default, Jun 22 2015, 17:58:13) [GCC 4.8.2]
  ntc@ntc:~/testing$
```

   Вы также увидите строку `library =`. В этой строке нужно добавить каталог, в котором сохранены все клонированные репозитории (например, `library = /etc/ntc/ansible/`). При правильном обновлении файла конфигурации при следующем выполнении команды `ansible --version` вы увидите все внесенные изменения.
5. Если сохраняется необходимость установки каких-либо пакетов ПО, от которых зависят модули, то эта процедура должна быть документирована на сайте каждого соответствующего проекта GitHub. При этом, возможно, потребуется установка нескольких пакетов с помощью утилиты pip. Примечание: если вы используете Python virtualenvs или в вашей системе установлено несколько версий Python, то потребуется определение значения переменной `ansible_python_interpreter` в среде Ansible.

Резюме по системе Ansible

Ansible – вполне надежная и гибкая система, с помощью которой можно решать разнообразные задачи автоматизации сети: от проверок согласованно-

Tlgm: @it_boooks

сти и формирования отчетов до более общих комплексных задач управления конфигурацией и процесса автоматизации. Архитектура без использования агентов, заложенная в основу Ansible, снижает барьер сложности для тех, кто начинает осваивать методики автоматизации сети. Следует особо отметить, что в этом разделе рассматривались лишь основы начального уровня освоения Ansible и приводились лишь простые примеры того, что можно сделать с помощью этого инструментального средства. Более подробную информацию о практическом использовании Ansible можно найти на сайте http://docs.ansible.com.

В следующем разделе будет рассматриваться Salt – инструментальное средство, применяющее другой подход к автоматизации сети.

Автоматизация сети с использованием Salt

Salt – это надежная и эффективная программная рабочая среда (framework), спроектированная и реализованная как чрезвычайно быстрый и простой в использовании канал обмена данными (дословно: «шина» – bus), предлагающий такие функциональные возможности, как автоматизированное управление конфигурацией, полная поддержка облачной среды, автоматизация сети, а также автоматизация, управляемая событиями, которая обеспечивает поддержку выполнения сетевых операций современного уровня в любой инфраструктуре.

Salt является весьма гибкой системой, которая позволяет автоматизировать широкий диапазон разнообразных типов сетевых устройств и компонентов, впрочем, как и другие инструментальные средства, рассматриваемые в текущей главе. Несмотря на некоторый ощутимый уровень сложности, Salt может быть установлен и настроен буквально в течение нескольких минут, после чего сразу же можно приступить к процедурам автоматизации сетевых устройств.

Как и в предыдущем разделе, посвященном Ansible, наша главная цель – предоставить читателю объем информации, достаточный для того, чтобы быстро начать практическую работу и сразу же использовать Salt для автоматизации обобщенных сетевых задач. Поэтому текущий раздел содержит шесть основных тем (подразделов):

- основы архитектуры Salt;
- общая информация о Salt;
- управление сетевыми конфигурациями с помощью Salt;
- выполнение функций Salt в удаленном режиме;
- инфраструктура Salt, управляемая событиями;
- некоторые другие функциональные возможности Salt.

Основы архитектуры Salt

С точки зрения архитектуры проектное решение Salt представляет собой простое ядро с динамически подключаемыми интерфейсами. В текущем разделе будет наглядно показано, что буквально все в Salt подключается и отключается

Tlgm: @it_boooks

динамически и с легкостью расширяется, включая процессы создания драйверов новых устройств для автоматизации сетевых устройств, использующих разнообразные API. Именно поэтому Salt можно применять для автоматизации любых типов сетевых устройств.

Изначально разработка Salt была основана на архитектуре с использованием агентов, которая не вполне подходила для автоматизации сети, поскольку, как нам уже известно, далеко не всегда имеется возможность без затруднений загрузить программные агенты на все типы сетевых устройств. В действительности на обычном сетевом оборудовании это сделать очень сложно, а иногда невозможно. С учетом повышенного спроса на решения по автоматизации без применения агентов архитектура Salt была усовершенствована, после чего стала возможной поддержка двух вариантов решений: с использованием агентов и без них. Кроме того, в обоих вариантах развертывания Salt поддерживает режим автоматизации сети, управляемой событиями, который будет рассматриваться ниже в текущем разделе.

Ядро Salt при настройке по умолчанию основано на архитектуре «колеса», в котором можно выделить центральную ось и спицы (hub-and-spoke architecture). Центр (hub), или центральный сервер, обозначается как *Salt master* (в программном обеспечении используется имя `salt-master`) и управляет «спицами» колеса или периферийными узлами, которые названы *Salt minions* (в программном обеспечении используется имя `salt-minion`) и по существу являются автоматизируемыми узлами. Координатор (Salt master) способен управлять тысячами миньонов (Salt minions). Обмен данными между координатором и миньонами осуществляется через постоянное соединение с использованием упрощенных протоколов, чтобы обеспечить передачу данных в реальном времени. Такой подход позволяет Salt быстро выполнять масштабирование и управлять более чем 30 000 миньонов при наличии единственного сервера-координатора. В решениях с бóльшим количеством узлов возможно распределение миньонов по нескольким серверам-координаторам, которые, в свою очередь, управляются координатором более высокого уровня.

Именно так работает Salt в обобщенном случае при автоматизации серверов. Чтобы понять, как работает Salt при автоматизации сетевых устройств, необходимо рассмотреть, как организована функциональность Salt при использовании архитектуры без агентов.

Использование архитектуры без агентов Salt с программным пакетом salt-ssh

Архитектура Salt была усовершенствована для работы в режиме без использования агентов. В этом режиме на автоматизируемых целевых узлах не устанавливается программный пакет `salt-minion`. Вместо него используется программный пакет `salt-ssh`, который можно установить непосредственно на сервере-координаторе или распределить по другим узлам, поскольку Salt предоставляет «шину» (канал) обмена данными между всеми процессами, связанными с Salt.

Tlgm: @it_boooks

При такой схеме решения координатор устанавливает соединение с целевым устройством по протоколу SSH, поэтому подобную архитектуру иногда сравнивают с архитектурой Ansible. Также следует отметить, что при использовании пакета `salt-ssh` остается полностью доступной вся функциональность Salt для автоматизации сетевой инфраструктуры.



Подсистема `salt-ssh` представляет собой всего лишь один из процессов, используемых в архитектуре Salt, и может быть установлена как на сервере-координаторе, так и на любой другой системе.

Но даже рабочий режим без применения агентов на основе программного пакета `salt-ssh` сам по себе не особенно полезен для процесса автоматизации сетевых устройств из-за наличия разнообразных типов транспортных протоколов, API и сетевых операционных систем. Причина заключается в недостатке комплексных объединяющих решений на основе SSH, созданных для Salt.

Поэтому в следующем подразделе мы рассмотрим наиболее эффективное решение для автоматизации сетевых устройств – использование в Salt прокси-миньонов.

Использование архитектуры Salt без агентов с прокси-миньонами

Еще одной методикой практической реализации процесса автоматизации без использования агентов является концепция *прокси-миньона* (*proxy minion*) в Salt. Прокси-миньоны представляют собой подмножество миньонов, следовательно, обладают всеми функциональными характеристиками и свойствами обычных миньонов. В сущности, прокси-миньон – это виртуальный миньон. Виртуальный миньон не устанавливается на автоматизируемом устройстве, он просто предоставляет доступ в режиме прокси к автоматизируемым устройствам. Прокси-миньоны обладают свойством расширяемости, предлагая возможности создания (или изменения) наиболее предпочтительного типа канала обмена данными от конкретного прокси-миньона к целевому автоматизируемому устройству. Именно так в настоящее время выполняется автоматизация сети с использованием Salt.



На управляемом устройстве или в миньоне создается процесс, связывающий его с прокси-миньоном. Для каждого такого процесса требуется 40 Мб оперативной памяти. При использовании прокси-архитектуры каждый прокси-миньон способен управлять 100 устройствами с прокси-компьютера, на котором доступно 4 Гб оперативной памяти (RAM). Такие характеристики вполне обосновывают выбор концепции прокси-миньонов для автоматизации сети. Процессы прокси управляются координатором и весьма часто выполняются на том же физическом сервере, но их можно также разместить и в распределенной архитектуре, улучшая тем самым способность Salt к масштабированию при управлении сетевыми устройствами. Например, с помощью Salt можно автоматизировать сеть, состоящую из 10 000 узлов, если распределить прокси-миньоны по 10 компьютерам. При этом на каждом сервере создается 1000 процессов прокси-миньонов, следовательно, каждый сервер управляет 1000 узлов.

Tlgm: @it_boooks

Автоматизация сетевых устройств с использованием Salt

Salt поддерживает автоматизацию сети, используя для этого прокси-миньоны. Для сетевой среды специально предназначены следующие четыре типа прокси-миньонов ядра:

- *NAPALM* – предлагает встроенную поддержку автоматизации сети, состоящей из устройств от различных производителей сетевого оборудования, с использованием библиотеки с открытым исходным кодом на языке Python (специально предназначенной для NAPALM), которая подробно рассматривается в приложении Б;
- *Cisco Network Services Orchestrator (NSO)* – коммерческое решение от компании Cisco, предлагающее архитектуру, управляемую моделью, для автоматизации сети, состоящей из устройств от различных производителей сетевого оборудования, с использованием в основном NETCONF;
- *Juniper* – используется для управления устройствами Juniper Junos. Разработка компании Juniper;
- *Cisco NX-OS* – используется для управления устройствами Cisco NXOS. Разработка SaltStack.

Во всех последующих примерах текущей главы все внимание сосредоточено на использовании прокси-миньона NAPALM для взаимодействия с разнообразными устройствами, в том числе с Cisco IOS, Cisco NXOS, Arista EOS и Juniper Junos. Такой выбор объясняется тем, что прокси-миньон NAPALM является весьма активно разрабатываемым ПО с открытым исходным кодом и обеспечивает широкую поддержку устройств от различных производителей сетевого оборудования.

Напомним, что для примеров текущего раздела используются устройства и сетевая топология, показанная на рис. 9.3 (воспроизводящем схему, показанную на рис. 9.1 в предыдущем разделе, в котором рассматривалась система Ansible).



Следует еще раз напомнить, что Salt обладает свойством расширяемости и наращивания функциональности в широких пределах, поэтому можно написать специализированные прокси-миньоны для различных устройств, которые используют особенные API или устаревшие интерфейсы, такие как SNMP или Telnet.

Общая информация о Salt

Для того чтобы приступить к освоению Salt, необходимо хорошо знать и понимать смысл терминов этой системы, а также основные концепции и их соответствие общей структуре программной среды Salt. Некоторые концепции мы рассмотрим более подробно. Начнем с файла формата SLS.

Файл формата SLS

Ранее в книге рассматривались шаблоны Jinja и файлы на языке YAML. В любом случае, Jinja – это всего лишь один из языков шаблонов, а YAML – лишь один способ структурирования данных и представления их в формате, очень

Tlgm: @it_boooks

удобном для чтения человеком. Представьте, что есть возможность использования одного файла, который понимает Jinja (и другие языки шаблонов) и YAML (и другие форматы данных), для формирования разнообразных наборов данных (то есть для вставки данных в шаблон). Такой файл действительно существует – это файл SLS.

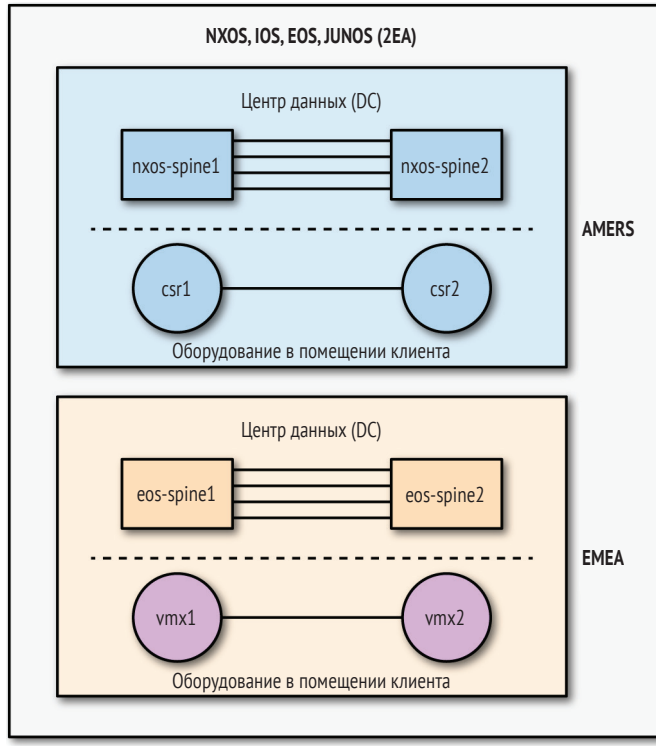


Рис. 9.3 ❖ Схема топологии сети

SLS – это специальный формат (SaLt State) файла для Salt. Это комбинация представления данных и языков шаблонов, которые можно объединить в одном и том же файле.

По умолчанию в SLS-файле используется комбинация YAML + Jinja. Но благодаря гибкости Salt и SLS можно с легкостью переключиться на любое другое сочетание. Это один из примеров высокой адаптируемости Salt к любым условиям с помощью подключения к ядру разнообразных внешних модулей: нет никаких факторов, ограничивающих выбор только Jinja и YAML, вы можете выбрать другие варианты. Например, для представления данных предлагаются следующие варианты: YAML, YAMLEX, JSON, JSON5, HJSON или даже сам язык Python, а для создания шаблонов выбор таков: Jinja, Mako, Genshi, Chee-

Tlgm: @it_boooks

tah, Wempy или тот же Python. Этот список можно расширить в соответствии с вашими требованиями и предпочтениями.

Одним из практических примеров поддержки различных типов представления данных и механизмов шаблонов является упрощение процедур перехода с других инструментальных средств на Salt. Например, если имеется собственный (или специализированный) инструмент, использующий шаблоны Mako (это тоже механизм шаблонов на основе языка Python), то можно без затруднений воспользоваться теми же шаблонами в среде Salt, поскольку использование Jinja не является обязательным условием.

Ниже приведен пример простейшего файла в формате SLS, написанный на языке YAML, как обычный файл данных:

```
ntp_peers:
  - 10.10.10.1
  - 10.10.10.2
  - 10.10.10.3
```

В этот файл можно добавить цикл языка Jinja `for`, для того чтобы сформировать более динамичную среду:

```
ntp_peers:
  {%- for peer_id in range(1, 4) %}
  - 10.10.10.{{ peer_id }}
  {%- endfor %}
```

Для демонстрации мощных возможностей файлов данных SLS при работе с другими форматами данных и типами шаблонов приводится следующий пример использования формата HJSON в сочетании с шаблоном Mako:

```
#!hjson|mako
ntp_peers: [
  % for peer_id in range(1, 4):
  ${peer_id},
  % endfor
]
```

Три примера, приведенных выше, представляют одни и те же данные – список трех пиринговых узлов NTP. В последнем примере используется связка HJSON и Mako – отметим наличие комбинации символов shebang в самой первой строке файла, позволяющей точно определить эту связку. HJSON – это расширение синтаксиса JSON, поддерживающее формат данных, более удобный для чтения человеком и снижающий потенциальную вероятность возникновения ошибок.

Как было отмечено выше, файлы данных SLS можно также создавать непосредственно на языке Python. Ниже приведен еще один пример представления тех же самых данных:

```
#!py
def run():
    return [
      '10.0.10.{}'.format(peer_id) for peer_id in range(1, 4)
    ]
```

Tlgm: @it_boooks

Во всех приведенных выше примерах показаны файлы SLS, но все они являются файлами данных. Подобные файлы содержат данные, которые в конечном итоге будут использоваться для выполнения задач автоматизации сети, такие как создание файлов конфигурации и конфигурирование устройств.

Обратите особое внимание на то, что рассматриваемые здесь файлы в формате SLS обязательно должны сохраняться с расширением *.sls*.

В следующем разделе рассматриваются распределительные pillar-интерфейсы и их взаимодействие с файлами данных в формате SLS.

Распределительные pillar-интерфейсы

Распределительный pillar-интерфейс представляет собой ресурс данных, который может быть SLS-файлом или данными, извлекаемыми из внешнего сервиса, например из CMDB или с другой платформы управления сетью.



При работе с pillar-файлами следует помнить, что главный файл конфигурации Salt, который будет рассматриваться в следующем разделе, обязательно должен содержать правильно определенные пути доступа к месту хранения pillar-файлов.

В pillar-файлах хранятся все данные, необходимые для управления сетевыми устройствами, в том числе и вся общая информация, как, например, регистрационные данные для процедуры аутентификации. Кроме того, в эти файлы включаются данные, требуемые для конфигурирования всех компонентов целевого устройства – от конфигурации интерфейса и протоколов до конфигурации BGP или NTP.

Ниже приведен пример pillar-файла с использованием формата файла SLS:

```
proxy:
  proxytype: napalm
  driver: nxos
  fqdn: nxos-spine1.dc.amers
  username: ntc
  password: ntc123
hostname: nxos-spine1
openconfig-bgp:
  bgp:
    global:
      config:
        as: 65001
        router_id: 172.17.17.1
```



В процессе разработки нашего учебного примера приведенный выше pillar-файл добавляется в основную управляющую конфигурацию. Затем он распределяется по всем прокси-миньонам, хотя в нашем случае имеется единственный прокси-миньон, установленный непосредственно на главном сервере-координаторе.

В приведенном выше pillar-файле определены три ключа. Имя первого ключа `proxy` – это специальное ключевое слово Salt, для которого требуется пара ключ-значение, устанавливающая связь с конкретным используемым прокси-миньоном. Другие ключи – `hostname` и `openconfig-bgp` – это произвольные

Tlgm: @it_boooks

выбранные по усмотрению пользователя ключи, определяющие значения данных, которые потребуются для процедуры конфигурирования и передачи соответствующих команд на сетевое устройство.

В рассматриваемом здесь примере pillar-интерфейс сохраняется в файле */etc/salt/pillar/nxos_spine1_pillar.sls*. Этот конкретный pillar-интерфейс является специализированным для определенного устройства, но далее в текущем разделе мы увидим, что pillar-интерфейс обладает более широкими возможностями и может хранить данные, используемые для набора из нескольких устройств.



Чтобы защитить секретные данные и не предъявлять их публично, можно зашифровать их с помощью программы GPG, при этом Salt будет расшифровывать эти данные динамически во время выполнения (https://docs.saltstack.com/en/latest/ref/renderers/all/salt.renderers.gpg.html). Другой вариант защиты – возможность сохранения секретных данных в надежно защищенном внешнем pillar-интерфейсе (например, Vault).



Для крупномасштабных разработок может потребоваться извлечение данных из некоторой внешней системы, которая уже существует в организации, а не управление многочисленными pillar-файлами. В таких случаях возможно применение внешних pillar-файлов. Такими внешними pillar-интерфейсами могут быть любые внешние сервисы, например базы данных, репозитории Git, HTTP API и даже таблицы Excel (приведенный список далеко не полон). Подробную справочную информацию по этой теме можно найти по адресу: https://docs.saltstack.com/en/latest/topics/development/external_pillars.html.

Обобщенный вариант автоматизации сети – извлечение данных из ПО IPAM (IP address management). С учетом того, что большинство решений IPAM предоставляет данные через API на основе HTTP, в pillar-файл можно добавить следующие три строки:

```
ext_pillar:
  - http_yaml:
      url: https://my-ipam.org/api/<node>
```

В этом случае все данные, возвращаемые из ПО IPAM, могут представлять собой данные, в той или иной степени используемые при выполнении задачи Salt, например при обработке данных в шаблоне, предназначенном для генерации конфигураций.

Top-файл

В предыдущих разделах были подробно описаны файлы формата SLS и pillar-файлы, использующие формат SLS. В Salt есть еще один тип файла, который использует формат SLS, – *top-файл*.

Top-файл, который часто называют просто *top*, определяет соответствие (связь) между миньоном или группой миньонов и данными (с использованием pillar-интерфейсов как посредников), с которыми должны работать эти миньоны. В определенной степени можно рассматривать top-файл как некоторый аналог inventory-файла в Ansible, но в действительности между ними существует много различий, которым мы уделим особое внимание.

В top-файле можно определить, какие pillar-интерфейсы назначаются для конкретных сетевых устройств. При добавлении в систему управления Salt

Tlgm: @it_boooks

новые устройства идентифицируются с помощью неповторяющегося идентификатора миньона (minion ID), присваиваемого пользователем. В дальнейшем можно ссылаться на этот идентификатор и устанавливать связь между определенными pillar-интерфейсами (то есть данными) и новым устройством или создавать расширенные группы на основе типа устройств, локации (места размещения устройств) или географического региона.

В общем случае top-файл определяется как top.sls. Для нашего примера top-файл сохранен как */srv/pillar/top.sls*.

Ниже приведен простой пример top-файла, в котором используются явно определяемые соответствия (связи) на основе идентификатора миньона, а также соответствия между каждым устройством и pillar-файлом данных.

```
base:
  csr1:  # идентификатор миньона
    - csr1_pillar  # для pillar-интерфейса устанавливается связь с csr1
  vmx1:
    - vmx1_pillar
  nxos-spine1:
    - nxos_spine1_pillar
  eos-spine1:
    - eos_spine1_pillar
```

В приведенном примере миньон с идентификатором `nxos-spine1` использует pillar-интерфейс *nxos_spine1_pillar.sls*.

Обратите внимание на ключевое слово `base`, указанное как корневой ключ в top-файле. В Salt `base` – это зарезервированное ключевое слово, определяющее среду по умолчанию, которая должна управляться данной системой Salt. Таким образом, с помощью системы Salt можно управлять несколькими средами, обозначенными различными ключами (prod, test, DR, QA) в top-файле. Во всех примерах текущего раздела используется среда по умолчанию, то есть base.

Как было отмечено выше, может потребоваться установление соответствия одного pillar-файла, содержащего конкретные данные конфигурации, с определенным типом устройств, то есть с несколькими устройствами. В этом случае идентификатор миньона не используется. Можно воспользоваться более эффективными методами, такими как шаблоны, аналогичные используемым в командных оболочках, и регулярные выражения, или даже использовать характеристики устройства, включающие grains-интерфейсы (данные), которые будут рассматриваться в следующем подразделе, или pillar-данные.

Рассмотрим еще несколько примеров top-файлов, несколько усложненных и более приближенных к практике, в которых используются шаблоны, аналогичные применяемым в командных оболочках, и регулярные выражения.

В следующем примере устанавливается связь pillar-данных с устройствами с использованием характеристик устройства, обозначаемых как grains-данные в системе Salt, в том числе данные о производителе оборудования и версии ОС:

Tlgm: @it_boooks


```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
```

В приведенном примере pillar-файл *junos.sls* загружается только для устройств, которые идентифицированы как произведенные компанией Juniper с помощью характеристики `vendor`. Еще раз напомним, что эти характеристики обозначены термином *grains* и мы рассмотрим их в следующем разделе. В примере grains-данные обозначаются комбинацией символов `G@`. Для pillar-файла *ios_16.sls* установлено соответствие всем устройствам IOS версии 16.X (только на этих устройствах будет загружаться указанный pillar-файл). Далее можно видеть, что pillar-данные spine загружаются для каждого устройства типа spine (например, `nxos-spine1` или `eos-spine2`). В этом примере используются регулярные выражения – обратите внимание на комбинацию символов `E@` (выражение). Искомый идентификатор миньона должен содержать любые символы (`.*`), за которыми обязательно следует литерал `-spine`, после которого должна обязательно присутствовать одна цифра (`\d`). Такое выражение полностью соответствует устройствам типа spine в нашей топологии.

Также можно определить pillar-данные по умолчанию, которые необходимо применять ко всем устройствам. Например, для загрузки ранее созданного pillar-файла *ntp_peers.sls* в top-файл можно добавить следующие строки:

```
  '*':
    - ntp_peers
```

Теперь мы можем быть вполне уверенными в том, что вся сеть будет использовать один и тот же набор пиринговых узлов NTP.

Кроме того, можно определить специальные группы на основе собственной бизнес-логики. Для установления связи с такой специальной группой устройств, идентифицируемой именем, присвоенным пользователем (еще немного аналогии с группами, определяемыми в Ansible), необходимо использовать ключ `nodegroups` в главном файле конфигурации Salt, то есть файле */etc/salt/master*:

```
nodegroups:
  amers:
    - 'csr*'
    - 'or'
    - 'nxos-spine*'
  emea:
    - 'vmx* and G@os:junos'
    - 'or'
    - 'eos-spine*'
```

Tlgm: @it_boooks



Напомним, что главный файл конфигурации Salt будет подробно рассматриваться в одном из следующих разделов.

Теперь определены группы `amers` и `emea`. Группа `amers` объединяет все устройства, идентификатор миньона которых начинается с подстроки *csr* или *nxos-spine*, а в группу `emea` включены устройства с начальной частью идентификатора *eos-spine* или устройства под управлением Junos, идентификатор которых начинается с символов *vmx*.

После определения этих групп в главном файле конфигурации на них можно ссылаться в top-файле. В следующем примере обратите внимание на два новых ключа: `N@emea` и `N@amers`. Они ссылаются на группы узлов (`N@`), только что определенные в главном файле конфигурации.

```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
  'N@emea':
    - communities_emea
  'N@amers':
    - communities_amers
```

Предполагается, что уже были созданы два pillar-файла для BGP-сообществ: *communities_amers.sls* и *communities_emea.sls*.



Не забывайте, что для top-файла по умолчанию остается принятым формат SLS, то есть комбинация Jinja+YAML, которую можно использовать для генерации динамических соответствий. Например, если имеется более длинный список регионов, то предыдущий пример мог бы выглядеть следующим образом:

```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
  {% for region in ['emea', 'amers', 'apac'] -%}
  'N@{{ region }}':
    - communities_{{ region }}
  {% endfor -%}
```



В этом разделе главное внимание сосредоточено на начальном уровне изучения Salt, тем не менее вам следует знать, как можно объединить Salt с внешними системами, предлагающими более динамичные top-функции. Вместо обычного top-файла можно воспользоваться внешним сервисом. Это очень удобно, если уже имеются inventory-файл и определения групп в какой-либо другой системе или инструментальной среде.

Tlgm: @it_boooks

Grains-интерфейсы (данные)

Выше уже упоминались объекты типа grains, но в этом подразделе мы рассмотрим их более подробно. Напомним, что pillar-интерфейсы уже определены в SLS-файлах. Таким образом, в нашем примере pillar-данные – это данные, предоставляемые пользователем. В отличие от них, grains-интерфейсы представляют данные, собираемые системой Salt.

Grains-данные – это информация, которую Salt собирает о конкретном устройстве, например производитель устройства, модель, серийный номер, версия ОС, ядро, DNS, дисковые подсистемы, процессоры и время непрерывной работы (uptime). Нет необходимости как-либо обрабатывать эти данные, но следует знать об их существовании, поскольку такие данные можно использовать разнообразными способами. Выше уже был продемонстрирован пример использования grains-данных в top-файлах. Но эти данные также можно использовать в шаблонах, в условных выражениях и в отчетах.



Grains-данные (интерфейсы) – это термин, применяемый только в системе Salt. В других инструментальных средствах этот тип данных часто обозначают термином facts (факты). Но следует отметить, что эти термины не вполне равнозначны – grains представляют собой абсолютно статические данные, которые кешируются. Динамически изменяющиеся подробности, такие как некоторые характеристики интерфейсов, конфигурация BGP, соседние устройства, определяемые по протоколу LLDP, извлекаются во время выполнения рабочего процесса через выполняемые модули (execution modules) Salt.

Кроме того, имеется возможность создавать собственные grains-интерфейсы, используя для этого специализированные компоненты интеграции в систему Salt в форме выполняемых модулей (execution modules) или с помощью статического определения grains-интерфейсов в файлах. Один из вариантов статического хранения grains-данных в файле конфигурации прокси-миньона показан ниже:

```
grains:
  role: spine
  production: true
```



Перед добавлением нового grains-интерфейса рекомендуется предварительно оценить уровень динамичности изменения представляемой информации. Grains-интерфейсы больше подходят для данных, вероятность изменения которых минимальна. Поэтому хранение данных в pillar-файлах является более предпочтительным вариантом.

В следующем разделе рассматривается использование выполняемых модулей для просмотра grains-данных, полученных от одного или нескольких устройств, с помощью команды `salt`.

Выполняемые модули

В Salt выполняемые модули (execution modules), которые чаще всего называют просто модулями, используются для извлечения данных, хранящихся под управлением системы Salt, или данных, получаемых непосредственно от устройства.

Tlgm: @it_boooks



В данной книге рассматривается лишь небольшая часть модулей. Полный список модулей можно найти на странице https://docs.saltstack.com/en/latest/salt-modindex.html.

Сначала рассмотрим наиболее часто используемые модули и соответствующие функции, применяемые для просмотра и grains-данных, и pillar-данных. Просмотр выполняется с помощью команды `salt`, но в дальнейшем вы увидите, что можно использовать эти модули непосредственно в SLS-файлах, применяемых для шаблонов, pillar-интерфейсов и прочих программных компонентов в системе Salt.

В первом примере просто выводятся grains-данные `model` для устройства `csr1` с помощью функции `get`, расположенной в выполняемом модуле `grains`.

```
$ sudo salt csr1 grains.get model
csr1:
    CSR1000V
```

Чтобы вывести полный список grains-данных, доступных для какого-либо миньона, необходимо воспользоваться функцией `grains.items` без передачи аргументов.

Также можно получить доступ к требуемым данным из pillar-файла, выполнив `salt` в командной строке. В следующем примере визуально проверяется вывод значения `ntp_peers` только для устройства `csr1`. Как и `grains.get`, функция `pillar.get` возвращает значение указанных pillar-данных.

```
$ sudo salt csr1 pillar.get ntp_peers
csr1:
    - 10.10.10.1
    - 10.10.10.2
    - 10.10.10.3
```

Чтобы извлечь значения из более сложной структуры данных, определенной в pillar-файле, необходимо воспользоваться разделителем : (двоеточие) для перемещения по иерархическим уровням ключ-значение. Обратившись к ранее определенной структуре из файла *nxos_spine1_pillar.sls*, которая для удобства еще раз показана здесь, можно вывести только значение BGP ASN для `nxos-spine1`:

```
# пример объекта в файле pillar-данных: nxos_spine1_pillar.sls
openconfig-bgp:
  bgp:
    global:
      config:
        as: 65001
        router_id: 172.17.17.1
```

```
$ sudo salt nxos-spine1 pillar.get openconfig-bgp:bgp:global:config:as
nxos-spine1:
    65001
```



Поскольку мы начали пользоваться командой `salt`, необходимо определить ее обобщенный синтаксис:

Tlgm: @it_boooks


```
$ sudo salt [options] <target> <function> [arguments]
```

Здесь `target` используется для определения миньонов, которые должны автоматизироваться с помощью заданных аргументов `arguments`.

Дополнительную информацию по использованию команды `salt` можно получить с помощью ключа `--help`, то есть выполнив команду `salt --help`. В текущем разделе мы продолжим рассмотрение примеров практического применения команды `salt`.

Сбор данных об устройствах с помощью сетевых модулей

В предыдущих примерах использовались модули `grains` и `pillar`. С помощью этих модулей легко получить доступ к предварительно определенным данным или к кешируемым данным, которые были собраны ранее.

В Salt также есть более десятка модулей для извлечения данных особых типов, связанных с функциональными характеристиками сетевых устройств, в том числе конфигурация NTP, пиринговые узлы NTP, BGP, таблица маршрутов, SNMP, список пользователей и многие другие данные. Существуют даже модули с еще более расширенной функциональностью для извлечения данных из устройств и представления их в формате YAML в виде, соответствующем моделям YANG.



Следует помнить, что вся описанная выше функциональность, например извлечение grains-данных или файлов конфигурации из устройств, хотя и предъявляется системой Salt, в действительности реализована с помощью Python-библиотеки NAPALM.

Ниже приведены примеры практического использования модулей непосредственно из командной строки.

Извлечение таблицы ARP из устройства с идентификатором миньона `csr1`:

```
$ sudo salt csr1 net.arp
# вывод не показан в целях экономии места
```

Извлечение таблицы MAC-адресов из устройства с идентификатором миньона vmx1:

```
$ sudo salt vmx1 net.mac
# вывод не показан в целях экономии места
```

Извлечение статистических данных NTP из устройства с идентификатором миньона vmx1:

```
$ sudo salt vmx1 ntp.stats
# вывод не показан в целях экономии места
```

Извлечение активных соседних узлов BGP из устройства с идентификатором миньона vmx1:

```
$ sudo salt vmx1 bgp.neighbors
# вывод не показан в целях экономии места
```

Извлечение конфигурации BGP из устройства с идентификатором миньона eos-spine1:

Tlgm: @it_boooks


```
$ sudo salt eos-spine1 bgp.config
# вывод не показан в целях экономии места
```



В приведенных выше примерах специализированные сетевые функции запускаются внутри соответствующего выполняемого модуля. В данном случае `net`, `ntp` и `bgp` – это имена выполняемых модулей, за которыми следуют имена функций, размещенных в модулях (например, `bgp.neighbors` и `bgp.config`).

Указание цели и комплексное соответствие

В предыдущих примерах выполнялась автоматизация только одного устройства. В Salt предлагается возможность использования механизма *указания цели* (*targeting*) для автоматизации нескольких устройств. Сам по себе механизм указания цели весьма прост, но может усложняться в соответствии с требованиями бизнес-логики. Рассмотрим несколько примеров.

Воспользовавшись флагом командной строки `-L`, можно явно определить список целевых устройств:

```
$ sudo salt -L csr1,vmx1,nxos-spine1 net.mac
```

Используя механизм шаблонов (подобный применяемому в командных оболочках), можно формировать выражения, содержащие символы-шаблоны:

```
$ sudo salt 'vmx*' ntp.stats
```

Кроме того, можно воспользоваться grains-данными с помощью флага `-G` и на основе этих данных указать целевые устройства.

```
$ sudo salt -G 'os:junos' bgp.neighbors
```

Можно даже определить требуемые целевые устройства с помощью их статических pillar-данных, используя для этого флаг `-I`:

```
$ sudo salt -I 'bgp:as_number:65512' bgp.config
```

Чтобы последний пример работал, требуется наличие данных конфигурации BGP в форме следующей эквивалентной структуры YAML, определенной в pillar-файле:

```
bgp:
  as_number: 65512
```

Теперь, когда понятно, как автоматизировать один миньон или группу миньонов с применением разнообразных флагов и опций, необходимо изучить механизм комплексного соответствия. *Комплексное соответствие* (*compound matching*) позволяет реализовать логику с использованием условных выражений, то есть сделать более гибким механизм указания целевых устройств, подлежащих автоматизации.

Для извлечения конфигурации BGP из всех миньонов, идентификаторы которых начинаются с символов `vmx`, работают под управлением ОС версии 15.1x и содержат предварительно определенный номер ASN в соответствующем pillar-файле, можно использовать следующую команду:

Tlgm: @it_boooks


```
$ sudo salt -C 'vmx* and G@version:15.1* and I@bgp:as_number:65512' bgp.config
```

Применяя механизм комплексного сответствия, можно воспользоваться флагом `-C`, затем указывать другие флаги, описанные выше, с помощью комбинации требуемого флага и символа `@`.

Со временем выражения комплексного соответствия могут существенно усложняться, поэтому их определения можно сохранять в главном файле конфигурации под тегом `nodegroups`:

```
nodegroups:
  vmx-15-bgp: 'vmx* and G@os:junos and G@version:15.1* and I@bgp:as_number:65512'
```

Кроме того, можно получить доступ и ссылаться на группу узлов прямо из командной строки:

```
$ sudo salt -N vmx-15-bgp bgp.config
```

Проверка активного состояния миньонов с помощью модуля test

В проекте разработки любого масштаба чрезвычайно важным этапом выявления и устранения проблем является процедура проверки текущей активности миньонов и их полной функциональной готовности. Эта процедура выполняется с помощью модуля `test`, точнее – с помощью функции `test.ping`.

```
$ sudo salt vmx1 test.ping
vmx1:
    True
```

В действительности `test.ping` – простая функция, возвращающая логическое значение `True`. Она используется для проверки того факта, что миньон действительно активен (в рабочем состоянии) и принят (зарегистрирован) координатором. Примечание: эта функция не является аналогом ICMP-утилиты ping.

Просмотр модуля и строк документации функций

На начальном этапе изучения системы Salt пользователь не осведомлен обо всех способах и тонкостях практического применения конкретных функций из какого-либо модуля. В этом случае можно воспользоваться опцией `sys.doc` в выполняемой команде. Указание `sys.doc` без аргументов выводит документацию по всем модулям.

Кроме того, можно обеспечить вывод строк документации (docstring) для конкретного модуля или функции:

```
$ sudo salt vmx1 sys.doc test.ping
test.ping:

    Used to make sure the minion is up and responding. Not an ICMP ping.

    Returns `True`.

    CLI Example:

        salt '*' test.ping
```

Tlgm: @it_boooks

Различные ключи и опции вывода информации для модулей

В команде `salt` допускается использование значительного количества ключей и опций. Наиболее часто применяется ключ `--out`, определяющий вывод информации в заданном формате. По умолчанию структура данных выводится в командной строке в виде, удобном для чтения человеком, с цветовой разметкой. Этот формат называется `nested`:

```
$ sudo salt vmx1 ntp.peers
vmx1:
    ----------
    comment:
    out:
        - 1.2.3.4
        - 5.6.7.8
    result:
        True
```

С помощью ключа `--out` можно вывести структуру данных в формате YAML, в формате JSON или даже в виде таблицы:

```
$ sudo salt --out=json vmx1 net.arp
{
    "vmx1": {
        "comment": "",
        "result": true,
        "out": [
            {
                "interface": "fxp0.0",
                "ip": "10.0.0.2",
                "mac": "2C:C2:60:FF:00:5F",
                "age": 1424.0
            },
            {
                "interface": "em1.0",
                "ip": "128.0.0.16",
                "mac": "2C:C2:60:64:28:01",
                "age": null
            }
        ]
    }
}
```

Ниже показан пример вывода данных в таблице:

```
$ sudo salt --out=table vmx1 net.arp
vmx1:
----------
    comment:
    ----------
    out:
    ----------
        ------------------------------------------------------
```

Tlgm: @it_boooks


```
        |  Age  | Interface |     Ip     |        Mac        |
        -----------------------------------------------------
        | 991.0 |   fxp0.0  |  10.0.0.2  | 2C:C2:60:FF:00:5F |
        -----------------------------------------------------
        |  None |   em1.0   | 128.0.0.16 | 2C:C2:60:64:28:01 |
        -----------------------------------------------------
    result:
    ----------
```

Кроме нескольких перечисленных выше доступных типов вывода информации, можно добавить и многие другие форматы. Несмотря на то что ключ `--out` предназначен для командной строки, имеется возможность взять за образец формат вывода, показанный на экране, и многократно использовать его в том же виде в различных сервисах, в том числе и для передачи данных во внешний сервис.

Передача (и возврат) данных во внешние сервисы рассматривается в следующем разделе.

Передача данных во внешние сервисы

Как вы уже убедились, данные с легкостью извлекаются и просматриваются в командной строке. Но иногда возникает необходимость передачи данных во внешний сервис. Используя ключ `--return`, можно определить, куда нужно передать данные, но при этом требуется точно задать имя адресата (returner).

Список доступных адресатов может быть различным, но наиболее часто используются имена Slack, HipChat, Redis, SMS, SMTP, Kafka и MySQL.

Например, можно сконфигурировать адресат Slack, добавив его имя в миньон, прокси-миньон или в главный файл конфигурации:

```
slack.channel: Network Automation
slack.api_key: d4735e3a265e16eee03f59718b
slack.username: salt
```

После этого становится возможным выполнение команды с флагом `--return` и с использованием стандартного потока вывода `stdout` для передачи данных в Slack. Ниже приведен пример такой команды:

```
$ sudo salt --return slack test.ping
# вывод не показан в целях экономии места
```



При выполнении команды сохраняется вариант вывода результата в командной строке в требуемой форме, но в то же время результат перенаправляется в заданный сервис. Если вывод в командной строке не нужен, то можно воспользоваться специальным значением ключа `--out=quiet`.

Определение источников выводимых данных имеет смысл только в командной строке, но указываемые адресаты могут использоваться и в других приложениях. Например, при работе с планировщиком (scheduler) Salt можно выполнять некоторые рабочие задачи в заданные интервалы времени, а вы-

Tlgm: @it_boooks

вод результатов передавать конкретно определенному адресату. Подобным образом при выполнении задачи в результате срабатывания некоторого триггера может потребоваться просмотр полученных в итоге данных в каком-либо средстве мониторинга.

Состояния Salt, состояния SLS-файлов и модули состояния

Состояния (states) Salt – это модули, используемые для управления, сопровождения и ввода в эксплуатацию конфигурации. Эти модули являются декларативным или императивным представлением системной конфигурации. Располагая источником истинных данных в pillar-файле, модуль состояния сравнивает эти данные с текущей конфигурацией, затем принимает решение: что необходимо удалить или добавить. С учетом способности современных сетевых устройств применять атомарные (или частичные) конфигурации эта задача выполняется еще проще. Нужно только сгенерировать предполагаемую конфигурацию и позволить устройству самому определить различия. В тех случаях, когда устройство не обладает такими функциональными возможностями, потребуется еще одна дополнительная операция, показанная в предыдущем примере.

Состояние SLS-файла Состояние SLS-файла (state SLS) – это дескриптор, определяющий, какие модули состояний будут выполнены в процессе применения определяемого состояния. Каждое состояние идентифицируется по имени `state_name`, определяемому пользователем, и вызывает функцию состояния, встроенную в Salt, с передачей списка аргументов.

```
<state_name>:
  <state_function>:
    - list of state arguments
```

Чтобы начать работу с состояниями SLS-файлов, необходимо запомнить следующие объекты:

- `state_name` – произвольно выбираемое имя состояния;
- `state_function` – функция состояния, которую необходимо выполнить.



Не следует путать состояние SLS-файла с модулем состояния: модуль состояния – это модуль на языке Python, который обрабатывает аргументы, выполняет программный код и выдает результат. Состояние SLS-файла вызывает одну или несколько функций состояния.

Модули состояния для автоматизации сети Для решения задач автоматизации сети существует несколько доступных модулей состояния: `netyang`, `netconfig`, `netacl`, `netntp`, `netsnmp`, `netusers`.

Одним из наиболее гибких и универсальных модулей является `netconfig`, который управляет и развертывает конфигурации на сетевых устройствах с использованием разнообразных шаблонов, определяемых пользователем. Поэтому главное внимание в текущем подразделе будет уделено именно модулю `netconfig`.

Tlgm: @it_boooks

В следующем примере используется функция состояния `netconfig.managed`:

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - debug: true
```

Здесь необходимо сделать краткий обзор различных вариантов использования термина `state` в файле из предыдущего примера:

- сам файл представляет собой состояние SLS-файла;
- `ntp_peers_example` – имя определяемого состояния;
- `netconfig` – встроенный модуль состояния, который управляет конфигурациями;
- `managed` или `netconfig.managed` – встроенная функция, являющаяся частью модуля состояния `netconfig`, специально предназначенная для развертывания конфигураций на сетевых устройствах.

Также напомним, что *ntp_template.j2* – это шаблон, который может использовать функциональность SLS-файла, а для данных, передаваемых в шаблон, источником могут служить pillar-файлы данных, которые тоже являются SLS-файлами. Здесь самое главное – помнить о том, что SLS – это тип файла.

Обновление главного файла конфигурации

Мы уже неоднократно упоминали разнообразные файлы, используемые в системе Salt, такие как pillar-файлы или шаблоны. Файлы этих типов необходимо хранить в конкретно определенных локациях на сервере-координаторе. Такие локации определяются в главном файле конфигурации.

Главный файл конфигурации (master configuration file) – это файл в формате YAML, который предварительно сконфигурирован с помощью параметров и опций, принимаемых по умолчанию, сразу после установки системы Salt. По умолчанию путь для хранения главного файла конфигурации определяется как */etc/salt/master* или */srv/master*.

Список параметров и опций, определяемых в процессе конфигурирования главного файла конфигурации, чрезвычайно велик, но из этого списка следует выделить два наиболее важных параметра: `file_roots` и `pillar_roots`. Это ключи файла конфигурации – напомним, что основой этого файла является язык YAML.

С помощью ключа `file_roots` на сервере-координаторе определяются локальные пути к локациям хранения разнообразных файлов: шаблонов, состояний, pillar-файлов и модулей расширений. Используемая структура достаточно гибка, для того чтобы позволить пользователю сформировать различные рабочие среды на одном компьютере (например, эксплуатационная рабочая среда, тестовая среда и DR).

Ниже приведен небольшой фрагмент нашего учебного файла конфигурации, в котором определяются пути к файлам `file_roots`:

```
file_roots:
  base:
    - /etc/salt/pillar
```

Tlgm: @it_boooks


```
    - /etc/salt/templates
    - /etc/salt/states
    - /etc/salt/reactors
```

Отметим использование base – особого ключевого слова Salt, которое уже упоминалось в предыдущих разделах. Ключевое слово base предназначено для определения путей к файлам в рабочей среде, принимаемой по умолчанию (default). Напомним, что с помощью различных ключей можно определить несколько рабочих сред. Например, если необходимо определить среду с именем dev, используемую исключительно для разработки, то можно применить структуру, подобную приведенной ниже:

```
file_roots:
  dev:
    - /home/admin/pillar
    - /home/admin/states
```

Структура pillar_roots очень похожа на структуру file_roots, но она указывает на каталог, в котором хранятся pillar-файлы:

```
pillar_roots:
  base:
    - /etc/salt/pillar
```

В дальнейшем содержимое pillar_roots можно изменить для поддержки специальной среды разработки dev:

```
pillar_roots:
  dev:
    - /home/admin/pillar
```



Объявление шаблонов в одном общем каталоге, таком как /etc/salt/template, не является обязательным требованием. Возможны проектные решения, в которых шаблоны хранятся в соответствии с каждым состоянием, то есть сгруппированы более логично по характеристикам их применения и по устройствам, использующим их.

После того как сформирована основа конфигурации в главном файле, следующим этапом становится выполнение однотипных задач в прокси-миньоне, особенно при автоматизации сетевых устройств.

Обновление файла конфигурации миньона и прокси-миньона

Каждый миньон имеет собственный файл конфигурации. По умолчанию поддерживаются следующие пути к этому файлу: */etc/salt/minion* и */srv/minion*. Поскольку прокси-миньон является специальной формой обычного миньона, он наследует все характеристики (YAML-ключи), поддерживаемые конфигурацией миньона.

Файл конфигурации прокси-миньона хранится в каталоге */etc/salt/proxy* или в каталоге */srv/proxy* (в зависимости от ОС).

Мы не вносили каких-либо изменений или обновлений в файл конфигурации прокси-миньона, принятый по умолчанию.

Tlgm: @it_boooks

В следующем разделе будут рассматриваться несколько рабочих процессов (потоков), которые создают и развертывают сетевые конфигурации.

Управление сетевыми конфигурациями с помощью Salt

Мы уже убедились в том, насколько гибкими и универсальными являются SLS-файлы и что можно сделать с помощью команды `salt`, но не менее замечательной функциональной возможностью Salt является то, что любая функция, доступная в командной строке, также может быть выполнена внутри SLS-файлов, включая и шаблоны.

Далее будет подробно рассматриваться процесс автоматической генерации конфигураций с использованием шаблонов.

Доступ к данным из шаблонов

Существует множество доступных встроенных идентификаторов (очень похожих на переменные), таких как `grains`, `pillar`, `opts` (словари параметров конфигурации) или `env` (набор переменных среды). Их также можно использовать непосредственно в шаблоне подобно тому, как они применяются в командной строке (например, `grains.get os`). Такая возможность весьма полезна при первоначальном формировании шаблонов сетевых конфигураций, и это будет продемонстрировано в нескольких следующих примерах.

Доступ к выполнению функций осуществляется с помощью зарезервированного ключевого слова `salt`. В командной строке просто выполнялась команда `ntp.peers`, но внутри шаблона к аналогичной команде необходимо добавить префикс `salt`:

```
{%- set configured_peers = salt.ntp.peers()['out'] -%}
```

Переменная `configured_peers` создается в шаблоне с помощью команды языка шаблонов Jinja `set`. Это переменная языка Jinja, поэтому при выполнении данного шаблона ей будет присвоен список активных пиринговых узлов NTP, сконфигурированный на целевом устройстве с помощью выполняемого модуля ntp.

В Salt, как и во многих других инструментальных средствах, настоятельно рекомендуется перемещать сложный уровень шаблонов Jinja в реальные активные функции, так чтобы данные или текущую задачу можно было в конечном итоге многократно использовать в других приложениях. Такой подход создает значительные преимущества: шаблоны становятся более удобными для чтения, предоставляется возможность их многократного использования, а кроме того, это очень удобный способ ввода различных наборов данных в систему.



Специализированные выполняемые модули можно достаточно быстро написать на языке Python, после чего они автоматически распространяются по миньонам (или прокси-миньонам). В дальнейшем такие модули могут применяться в шаблонах, в других SLS-файлах или непосредственно в командной строке.

Tlgm: @it_boooks

При использовании в шаблоне ключевых слов `grains`, `pillar` и `opts` можно определить бизнес-логику высокого уровня и проектировать шаблоны, не зависящие от конкретных производителей оборудования, создавая возможность выполнения на различных платформах одного и того же шаблона, обладающего достаточным уровнем «интеллекта», чтобы определить, какие изменения конфигурации необходимо загрузить.

Создание шаблонов сетевой конфигурации на языке Jinja

Можно определить шаблон Salt для генерации конфигурации пиринговых узлов NTP с использованием входных данных из pillar-файла, определенного ранее с помощью шаблона Jinja, который выглядит следующим образом:

```
{%- if grains.os == 'junos' %}
system {
 replace:
ntp {
   {%- for peer in pillar.ntp_peers %}
   peer {{ peer }};
   {%- endfor %}
 }
}
{%- elif grains.vendor | lower == 'cisco' %}
no ntp
  {%- for peer in pillar.ntp_peers %}
ntp peer {{ peer }}
  {%- endfor %}
{%- endif %}
```

Особенностью этого шаблона является прямое использование ключевого слова `pillar`. Это позволяет получить доступ к данным, определенным в pillar-файлах.

Если бы целью являлось конфигурирование конкретной функциональной характеристики в полном соответствии с требованиями в декларативной форме (без сохранения дополнительных пиринговых узлов на целевом устройстве), то можно было бы выполнить операцию `replace` на целевом устройстве. На платформе Junos для этого применяется ключевое слово `replace`, тогда как на других платформах (ОС) требуется команда отрицания (команда `no`).



В приведенном выше примере ключевому слову `replace` для Junos поставлена в соответствие NETCONF-операция `replace`, которая рассматривалась в главе 7. Эта операция позволяет заменить полностью всю иерархию в конфигурации.

Шаблон из приведенного выше примера сохранен под именем *ntp_template.j2* в каталоге */etc/salt/templates* в соответствии с определением `file_roots` в главном файле конфигурации.

В дальнейшем к этому шаблону можно обращаться как `salt://ntp_template.j2` при использовании его в командной строке или в файлах состояния Salt.

Tlgm: @it_boooks

На этом этапе шаблон Jinja был только лишь сформирован, с его помощью пока еще не обрабатывались данные и не создавался какой-либо файл конфигурации.

Для демонстрации возможностей применения директивы `salt` в шаблоне можно определить список пиринговых узлов NTP, которые должны добавляться или удаляться на основе извлекаемых в реальном времени сведений о существующих пиринговых узлах с помощью команды `salt.ntp.peers`.

Следующий шаблон создает конфигурацию для платформ IOS и Junos с функцией конфигурирования пиринговых узлов NTP. В этом шаблоне описана логика, требуемая для гарантии того, что в конечном итоге на целевом устройстве будут сконфигурированы только пиринговые узлы, определенные в pillar-файле. Таким образом, все ненужные пиринговые узлы будут удалены с целевого устройства (при развертывании данной конфигурации).

```
{%- set configured_peers = salt.ntp.peers()['out'] -%}
{%- set add_peers = pillar.ntp_peers | difference(configured_peers) -%}
{%- set rem_peers = configured_peers | difference(pillar.ntp_peers) -%}
{%- if grains.os == 'junos' -%}
 {%- for peer in rem_peers -%}
delete system ntp peer {{ peer }}
 {% endfor -%}
 {%- for peer in add_peers -%}
set system ntp peer {{ peer }}
 {% endfor -%}
{%- elif grains.vendor | lower == 'cisco' %}
 {%- for peer in rem_peers -%}
no ntp peer {{ peer }}
 {% endfor -%}
 {%- for peer in add_peers -%}
ntp peer {{ peer }}
 {% endfor -%}
{%- endif -%}
```

Для таких устройств, как Juniper, которые обеспечивают поддержку частичной замены характеристик и опций конфигурации, это вполне приемлемое решение. В других случаях оно может показаться слишком трудоемким из-за необходимости определения логики для каждой команды отрицания `no`, предназначенной для удаления ненужных пиринговых узлов. Тем не менее это наилучший способ обработки всех вероятных сценариев, для которых не существует собственных подходящих методик частичной замены конфигурации с помощью операций `replace`.

Развертывание сетевых конфигураций с использованием модуля состояния netconfig

Теперь необходимо определить состояние, которое может быть применено для вставки данных из NTP pillar-файлов в шаблон NTP для генерации команд, передаваемых на целевые устройства.

Tlgm: @it_boooks

Для этой цели воспользуемся функцией состояния `netconf.managed`. Функция обрабатывает входные данные, создавая требуемую конфигурацию, и развертывает ее, выполняя соответствующие команды на сетевом устройстве.

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - debug: true
```

Этот SLS-файл состояния сохранен в одном из каталогов, определяемых `file_roots` (например, в каталоге */etc/salt/states*), под именем *ntp.sls*.

Для выполнения этого SLS-файла состояния необходимо вызвать функцию `state.apply` или `state.sls` с указанием имени файла состояния как аргумента:

```
$ sudo salt vmx1 state.apply ntp
vmx1:
----------
          ID: ntp_peer_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 10:48:16.160777
    Duration: 4331.08 ms
     Changes:
              ----------
              diff:
                  [edit system ntp]
                  +    peer 10.10.10.1;
                  +    peer 10.10.10.3;
                  +    peer 10.10.10.2;
                  -    peer 1.2.3.4;
                  -    peer 5.6.7.8;
              loaded_config:
                  delete system ntp peer 1.2.3.4
                  delete system ntp peer 5.6.7.8
                  set system ntp peer 10.10.10.1
                  set system ntp peer 10.10.10.3
                  set system ntp peer 10.10.10.2

Summary for vmx1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   4.331 s
```

Отметим, что при однократном выполнении команды были сгенерированы в оперативной памяти и развернуты на сетевом устройстве. В этом примере предварительно не был создан файл конфигурации.

Формат вывода информации в командной строке, показанный в приведенном примере, называется *high-state*, но возвращаемый объект остается объек-

Tlgm: @it_boooks

том языка Python (это можно проверить с помощью ключа `--out=raw`), следовательно, его можно использовать многократно для определения более сложных рабочих процессов (потоков).

Отметим в возвращенных данных наличие ключа `loaded_config`, поскольку в файле состояния был задан параметр `debug: true`. Ключ `loaded_config` содержит конфигурацию, сгенерированную в соответствии с требованиями бизнес-логики.

Время выполнения относительно невелико с учетом всех проделанных операций: извлечение текущей конфигурации, определение различий, генерация конфигурации (все это внутри шаблона) и последующая загрузка на устройство команд, сгенерированных по обнаруженным различиям, затем ввод созданной конфигурации в память целевого устройства. Все это заняло чуть более 4.3 секунды.

Кроме того, можно выполнить тот же самый файл состояния *ntp.sls* для данных `csr1`. Это состояние обрабатывает тот же шаблон, зная при этом из grains-данных, что `csr1` – это устройство Cisco IOS, поэтому будет сгенерирована соответствующая конфигурация для загрузки на заданное целевое устройство:

```
$ sudo salt csr1 state.apply ntp
csr1:
----------
          ID: ntp_peer_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 11:38:14.609398
    Duration: 3414.471 ms
     Changes:
              ----------
              diff:
                  + ntp peer 10.10.10.1
                  + ntp peer 10.10.10.3
                  + ntp peer 10.10.10.2
              loaded_config:
                  ntp peer 10.10.10.1
                  ntp peer 10.10.10.3
                  ntp peer 10.10.10.2

Summary for csr1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:  3.414 s
```

Использование зависимостей состояний

Еще одной важной функциональной возможностью, которой можно воспользоваться в файлах состояния, является создание зависимостей состояний.

Tlgm: @it_boooks

Когда необходимо применить несколько состояний, зависящих друг от друга, весьма полезными оказываются *реквизиты* (или *характеристики*) *состояния* (*state requisites*) (https://docs.saltstack.com/en/develop/ref/states/requisites.html). Например, если нужно выполнить состояние `ntp_peer_example` только в том случае, если было успешно выполнено другое состояние (как, например, `bgp_neighbors_example`), то достаточно добавить всего лишь две дополнительные строки:

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - require:
      - bgp_neighbors_example
```

Генерация файлов сетевой конфигурации

Кроме того, существует возможность разделения генерации конфигурации и ее развертывания на два отдельных этапа, подобно тому, как было показано в разделе об Ansible. Такой подход часто бывает удобен, если требуется проверка или просмотр нескольких версий наборов команд до начала процедуры реального развертывания.

Для этой цели используется функция состояния `file.managed`. В качестве аргументов в нее передаются тип шаблона и место его расположения. После обработки данных в шаблоне результат будет сохранен в файле *ntp_generated.conf* с использованием ключа `name`:

```
build_config:
  file.managed:
    - name: /home/admin/ntp_generated.conf
    - source: salt://ntp_template.j2
    - template: jinja
```

Если сохранить этот фрагмент в файле *build-ntp.sls*, то создание файла конфигурации можно будет выполнять следующей командой:

```
$ sudo salt csr1 state.apply build-ntp
```

Генерация и развертывание сетевых конфигураций из файлов

Если требуется создание конфигурации и ее развертывание в одном рабочем потоке, но при этом файл конфигурации сначала должен быть сгенерирован на сервере, то предлагается возможность объединения обоих этих состояний в одном SLS-файле.

```
generate_config:
  file.managed:
    - name: /home/admin/ntp_generated.conf
    - source: salt://ntp_template.j2
    - template: jinja
ntp_peer_example:
  netconfig.managed:
    - template_name: /home/admin/ntp_generated.conf
    - require:
      - file: /home/admin/ntp_generated.conf
```

Tlgm: @it_boooks

Если сохранить эти состояния в файле *ntp-build-deploy.sls* и выполнить этот файл, то получим следующий результат:

```
$ sudo salt vmx1 state.sls ntp-build-deploy
vmx1:
----------
          ID: /home/admin/ntp_generated.conf
    Function: file.managed
      Result: True
     Comment: File /home/admin/ntp_generated.conf updated
     Started: 12:17:25.544779
    Duration: 141.895 ms
     Changes:
              ----------
              diff:
                  ---
                  +++
                  @@ -0,0 +1,4 @@
                  + set system ntp peer 10.10.10.1
                  + set system ntp peer 10.10.10.3
                  + set system ntp peer 10.10.10.2
                  +
----------
          ID: ntp_peer_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 12:17:25.687279
    Duration: 4189.027 ms
     Changes:
              ----------
              diff:
                  [edit system]
                  +   ntp {
                  +       peer 10.10.10.1;
                  +       peer 10.10.10.3;
                  +       peer 10.10.10.2;
                  +   }
Summary for vmx1
------------
Succeeded: 2 (changed=2)
Failed:    0
------------
Total states run:     2
Total run time:   4.331 s
```

Параметризация имен файлов конфигурации

В двух предыдущих примерах при генерации файлов конфигурации использовалось имя файла */home/admin/ntp_generated.conf*. Такое решение не является гибким и масштабируемым, поскольку имя файла определено статически.

Tlgm: @it_boooks

Чтобы избежать неизменяемой кодировки имени файла и генерировать это имя в зависимости от конкретного устройства или идентификатора миньона, можно воспользоваться полем `id` из специальной SLS-переменной `opts`:

```
generate_config:
  file.managed:
    - name: /home/admin/{{ opts.id }}_ntp_generated.conf
    - source: salt://ntp_template.j2
    - template: jinja
```

Показанное здесь состояние генерирует файл */home/admin/vmx1_ntp_generate.conf* для миньона `vmx1`, файл */home/admin/csr1_ntp_generate.conf* для `csr1` и т. д.

Планирование выполнения состояний

В Salt очень важно различать рабочие задачи, выполняемые на сервере-координаторе, и рабочие задачи, выполняемые на миньоне. Миньоны запускают выполняемые функции, в то время как сервер-координатор выполняет так называемых «курьеров» (runners), которые будут рассматриваться в следующих подразделах. Это весьма существенное различие при планировании расписания выполнения заданий: если необходимо запланировать выполнение функции, инструкции добавляются в файл конфигурации миньона или прокси-миньона. При планировании работы курьера соответствующие параметры и опции добавляются в главный файл конфигурации. В обоих случаях синтаксис одинаков. Например, если требуется применение состояния из предыдущего примера в 11:00 каждого понедельника, нужно добавить следующие строки в файл конфигурации соответствующего (прокси) миньона:

```
schedule:
  ntp_state_weekly:
    function: state.sls
    args:
      - ntp
    kwargs:
      test: true
    ret: smtp
    when:
      - Monday 11:00am
```

В составе ключа `kwargs` сконфигурирован параметр `test: true`, то есть это состояние должно выполняться в тестовом (проверочном) режиме, но при этом возвращаются все различия в конфигурации. Более того, с помощью поля `ret: smtp` введен дополнительный, более тонкий параметр настройки, который определяет, что Salt должен принять вывод от этого состояния и перенаправить принятую информацию в *ретернер* (*returner*) с именем `smtp` (https://docs.saltstack.com/en/develop/ref/returners/all/salt.returners.smtp_return.html). Ретернер принимает данные, выводимые состоянием, и отправляет сообщение электронной почты, содержащее различия в конфигурации.

Tlgm: @it_boooks

Генерация отчетов

Генерация отчетов особенно полезна, когда эти отчеты активно используются рабочим процессом или человеком. Для этого наиболее удобны и просты в применении *ретернеры* (*returners*). В предыдущем примере выполнялось состояние NTP, затем выведенная им информация обрабатывалась с помощью ретернера SMTP (в основном выполняется передача отчета о выполнении в сообщении электронной почты).

Для передачи сообщения e-mail с сохранением содержимого в неизменном виде («как есть») необходимо сконфигурировать следующие параметры соответствующего миньона:

```
smtp.from: ping@mirceaulinic.net
smtp.to: jason@networktocode.com
smtp.host: localhost
smtp.subject: NTP state report
```

Содержимое (тело) сообщения можно дополнительно настроить с помощью шаблона, как показано ниже:

```
smtp.template: salt://ntp_state_report.j2
```

Здесь *ntp_state_report.j2* располагается в файловой системе Salt, например в каталоге */etc/salt/template*:

```
NTP consistency check
--------------------

Running on {{ id }}, which is a {{ grains.vendor }} {{ grains.model }} device,
running {{ grains.os }} {{ grains.version }}:

{{ result }}
```

При выполнении этой задачи планировщик отправляет сообщение электронной почты со следующим содержимым:

```
NTP consistency check
--------------------

Running on vmx1, which is a Juniper VMX device,
running junos 15.1F4.15:

vmx1:
----------
          ID: ntp_peer_example
    Function: netconfig.managed
      Result: None
     Comment: diff:
              [edit system]
              +   ntp {
              +       peer 10.10.10.1;
              +       peer 10.10.10.3;
              +       peer 10.10.10.2;
              +   }
```

Tlgm: @it_boooks


```
              Configuration discarded.
     Started: 15:14:09.911816
    Duration: 969.945 ms
     Changes:

Summary for vmx1
------------
Succeeded: 1 (unchanged=1)
Failed:    0
------------
Total states run:     1
Total run time: 969.945 ms
```

При такой настройке можно быть вполне уверенным в том, что Salt будет периодически выполнять состояние NTP в тестовом режиме, затем генерировать отчет о выполнении и отправлять его по электронной почте.

Чтобы достичь этой цели, можно вручную выполнить в командной строке следующую команду:

```
`$ sudo salt vmx1 state.sls ntp test=True --return smtp`
```

Аналогичным образом можно настроить одновременную передачу отчетов с результатами от многих устройств в определенное время, используя для этого курьера (runner) вместо выполняемой функции. Выполняемая функция запускается процессом миньона, тогда как функция-курьер активизируется процессом сервера-координатора, таким образом, обеспечивается полный обзор всей сети в целом. В терминах языка Python результат – это словарь, ключи которого соответствуют идентификаторам миньонов, а значения представляют текущие результаты выполнения состояния на каждом устройстве.

При доступности из командной строки и из процесса планировщика ретернеры являются весьма мощным инструментальным средством для обработки результатов выполнения и преобразования данных. В следующих разделах мы рассмотрим возможности многократного использования ретернеров для ответной реакции на события или для мониторинга работы всей системы Salt в целом.

Удаленное выполнение функций Salt

До настоящего момента мы рассмотрели лишь некоторые функциональные возможности Salt, но одним из наиболее важных компонентов, требующих полного понимания, является архитектура, реализованная в системе Salt для сетевых устройств. Как вам уже известно из предыдущего материала текущего раздела, предлагается два основных способа возможного взаимодействия с системой Salt и выполнения любых команд или задач в удаленном режиме с любого другого компьютера в сети. Существуют встроенные RESTful API и внешний пакет на языке Python `pepper`, который позволяет в удаленном режиме выполнять команды `salt` на сервере-координаторе.

Tlgm: @it_boooks

Использование Salt API

Собственный RESTful API включен в систему Salt, и его можно использовать для выполнения любых операций, которые возможно реализовать с помощью команд `salt`, как программы в командной оболочке Linux.

Главная функциональная характеристика этого RESTful API – он позволяет выбрать один из трех веб-серверов, поддерживаемых и доступных сразу после установки системы Salt: CherryPy, uWSGI или Tornado.

Ниже показано, как можно разрешить доступ к серверу CherryPy, отредактировав главный файл конфигурации:

```
rest_cherrypy:
  port: 8001
  ssl_crt: /etc/nginx/ssl/my_certificate.pem
  ssl_key: /etc/nginx/ssl/my_key.key
```

При этом определяется конфигурация сервера для прослушивания порта `8001` и использование сертификата и ключа для защищенных запросов.

Далее можно начать выполнение функций Salt в удаленном режиме, используя для этого специальные скрипты, Postman или cURL. В следующем примере показано применение cURL для извлечения таблицы ARP для `vmx1`.

```
curl -sSk https://salt-master-ns-or-ip:8001/run \
    -H 'Content-type: application/json' \
    -d '[{
        "client": "local",
        "tgt": "vmx1",
        "fun": "net.arp",
        "username": "ntc",
        "password": "ntc123",
        "eauth": "pam"
    }]'
```

В запросах, относящихся к конфигурации, функция заменяется на `state.sls` или `state.apply`, а имя требуемого состояния определяется в поле `args`:

```
curl -sSk https://salt-master-ns-or-ip:8001/run \
    -H 'Content-type: application/json' \
    -d '[{
        "client": "local",
        "tgt": "vmx1",
        "fun": "state.sls",
        "args": ["ntp"],
        "username": "ntc",
        "password": "ntc123",
        "eauth": "pam"
    }]'
```

При выполнении этого примера активизируется состояние NTP, определенное ранее в текущей главе.

Tlgm: @it_boooks

Salt и pepper

Использование встроенного API – не единственный возможный вариант, хотя наиболее предпочтительный, если необходимо объединить Salt с другими системами. Тем не менее существует и другая возможность – применение `pepper`, библиотеки на языке Python, позволяющая выполнять команды в CLI-интерфейсе как с обычного компьютера, так и непосредственно на сервере-координаторе Salt.

Библиотека `pepper` устанавливается с PyPI с помощью утилиты `pip` следующим образом: `pip install salt-pepper`.

Для работы библиотеки `pepper` требуется только конфигурирование регистрационных данных как переменных среды или в файле конфигурации *$HOME/.pepperrc*:

```
[main]
SALTAPI_URL=https://salt-master-ns-or-ip:8001/
SALTAPI_USER=my_username
SALTAPI_PASS=my_password
SALTAPI_EAUTH=pam
```

В библиотеку включен бинарный выполняемый файл для командной строки `pepper`, который можно использовать точно так же, как основную команду `salt`. Например, можно активизировать состояние NTP со своего компьютера, и оно будет выполнено непосредственно на сервере-координаторе:

```
$ pepper 'vmx1' state.sls ntp
# вывод не показан в целях экономии места
```

Управляемая событиями инфраструктура Salt

Система Salt построена на основе так называемой шины событий (event bus) (или канала событий), представляющей собой открытую систему на программной базе ZeroMQ, используемой для оповещения Salt и других систем о выполняемых операциях. ZeroMQ – это независимый от платформы высокопроизводительный инструментальный пакет поддержки обмена сообщениями в асинхронном режиме, позволяющий весьма эффективно выполнять рабочие задания без дополнительных издержек.

Для просмотра событий в реальном времени на сервере-координаторе выполняется следующая команда:

```
$ sudo salt-run state.event pretty=True
```

Если необходим контроль за модулем, используемым в команде `salt`, то при выполнении такой команды можно выделить три отдельных события. Например, команда `$ sudo salt -G os:nxos test.ping` генерирует следующие три события.

Во-первых, это идентификатор задачи Job ID, обеспечивающий единственную в своем роде ссылку на любое конкретное событие, связанное с целевыми миньонами. Это событие описывается следующим образом:

Tlgm: @it_boooks


```
20170619145155855122  {
    "_stamp": "2017-06-19T14:51:55.855336",
    "minions": [
        "nxos-spine1"
    ]
}
```

Во-вторых, сама задача, выполняемая на заданных миньонах. Описание этого события показано ниже:

```
salt/job/20170619145155855122/new {
    "_stamp": "2017-06-19T14:51:55.855656",
    "arg": [],
    "fun": "test.ping",
    "jid": "20170619145155855122",
    "minions": [
        "nxos-spine1"
    ],
    "tgt": "os:nxos",
    "tgt_type": "grain",
    "user": "sudo_admin"
}
```

Наконец, последнее событие для рассматриваемой команды – ответ и состояние для каждого миньона, включенного в целевую область видимости.

```
salt/job/20170619145155855122/ret/nxos-spine1 {
    "_stamp": "2017-06-19T14:51:55.867958",
    "cmd": "_return",
    "fun": "test.ping",
    "fun_args": [],
    "id": "nxos-spine1",
    "jid": "20170619145155855122",
    "retcode": 0,
    "return": true,
    "success": true
}
```

Отметим, что в последнем событии указан шаблон неповторяющегося тега для каждого миньона. Как вы можете видеть, в предыдущем примере был показан тег `salt/job/ 20170619145155855122/ret/nxos-spine1`.

В следующем разделе будут рассматриваться объекты, имеющие в системе Salt особенное значение для управляемой событиями автоматизации сети.

Наблюдение за внешними процессами с помощью маяков

В системе Salt маяки (beacons) используются для наблюдения за внешними процессами, которые не связаны напрямую с Salt, а также для импорта и возврата событий в шину Salt.

Например, маяк `inotify` используется для наблюдения за изменениями файла. Если необходимо отслеживать обновления файла *ntp_peers.sls*, то в файл

Tlgm: @it_boooks

конфигурации соответствующего (прокси) миньона нужно добавить следующие строки:

```
beacons:
  inotify:
    - /etc/salt/pillar/ntp_peers.sls:
        mask:
          - modify
      disable_during_state_run: True
```

Эти инструкции сообщают системе Salt о необходимости начать наблюдение за файлом */etc/salt/pillar/ntp_peers.sls* и передавать события в шину. При любом изменении содержимого этого файла мы увидим события, имеющие следующую структуру:

```
salt/beacon/vmx1/inotify//etc/salt/pillar/ntp_peers.sls {
    "_stamp": "2017-06-20T10:17:49.651695",
    "change": "IN_IGNORED",
    "id": "vmx1",
    "path": "/etc/salt/pillar/ntp_peers.sls"
}
```

Это может оказаться весьма полезным для отслеживания изменений различных данных в системе. Поскольку модули можно использовать в pillar-файлах (как в приведенном примере), данные являются динамическими и часто извлекаются из устройств в реальном времени. Использование маяков предоставляет возможность просмотра изменений в этих данных также в реальном времени.

Перенаправление событий с помощью механизмов

Специализированные механизмы (engines) представляют собой еще одну подсистему, взаимодействующую с шиной событий. Маяки только прослушивают внешние процессы и выполняют преобразование их в события Salt, в то время как механизмы могут работать в двунаправленном режиме. Основная задача механизмов – перенаправление событий, тем не менее они способны передавать сообщения непосредственно в шину. В этом и заключается главное различие между маяками и механизмами: маяки опрашивают конкретный сервис с заданным интервалом (по умолчанию 1 секунда), а механизмы могут обрабатывать и перенаправлять события сразу после их возникновения.

Очень полезным практическим применением механизмов может быть ведение журналов событий Salt на одном из syslog-серверов, например на Logstash, с использованием механизма *http-logstash*. В главном файле конфигурации это можно определить следующим образом:

```
engines:
  - http_logstash:
      url: https://logstash.elastic.co/salt
      tags:
        - salt/job/*/new
        - salt/job/*/ret/*
```

Tlgm: @it_boooks

Определение на языке YAML в главном файле конфигурации сообщает серверу-координатору о необходимости передачи событий на Logstash. Но в этом варианте указано, что передаваться должны только события, соответствующие тегам `salt/job/*/new` и `salt/job/*/ret/*`. Если бы ключ `tags` отсутствовал или оставался пустым, то механизм перенаправлял бы все события.

Прослушивание шины Salt с помощью реакторов

Система реакторов (reactor system) прослушивает шину событий и при возникновении события выполняет некоторое действие. Реакторы конфигурируются в главном файле на сервере-координаторе в соответствии с общим синтаксисом:

```
reactor:
  - <tag match>:
    - <list of SLS descriptors to execute>
```

Соответствие тегу (tag match) описывает шаблон, которому должен соответствовать тег события.

```
reactor:
  - 'salt/beacon/*/inotify//etc/salt/pillar/ntp_peers.sls':
    - salt://run_ntp_state_on_pillar_update.sls
```

В этом примере системе Salt сообщается о необходимости выполнения файла данных *run_ntp_state_on_pillar_update.sls*, когда маяк `inotify` передает в шину событие, соответствующее факту обновления отслеживаемого файла.

Реактор SLS *run_ntp_state_on_pillar_update.sls* может иметь любую структуру по усмотрению пользователя. Для рассматриваемого здесь примера принята следующая структура:

```
run_ntp_state:
  local.state.sls:
    - tgt: {{ data['id'] }}
    - arg:
      - ntp
    - ret: mongo
```

Здесь выполняется функция `state.sls` с аргументом `ntp` в применении к миньону, идентификатор которого извлекается из тела события, точнее из поля `id`.

Ниже приведены события, которые вы должны увидеть в шине событий. Сначала выводится pillar-файл *ntp_peers.sls*, содержимое которого изменилось.

```
salt/beacon/vmx1/inotify//etc/salt/pillar/ntp_peers.sls {
    "_stamp": "2017-06-20T10:57:24.651644",
    "change": "IN_IGNORED",
    "id": "vmx1",
    "path": "/etc/salt/pillar/ntp_peers.sls"
}
20170620105724736722  {
    "_stamp": "2017-06-20T10:57:24.737525",
```

Tlgm: @it_boooks


```
    "minions": [
        "vmx1"
    ]
}
salt/job/20170620105724736722/new {
    "_stamp": "2017-06-20T10:57:24.737804",
    "arg": [
        "ntp"
    ],
    "fun": "state.sls",
    "jid": "20170620105724736722",
    "minions": [
        "vmx1"
    ],
    "tgt": "vmx1",
    "tgt_type": "glob",
    "user": "sudo_admin"
}
# далее выводится результат выполнения состояния, который здесь не показан
# в целях экономии места
```

Первое событие сгенерировано маяком `inotify`, затем его подхватывает и перенаправляет реактор, который создает новую задачу и идентифицирует целевые миньоны. После этого созданная задача передается в идентифицированные миньоны (в данном случае это только один миньон `vmx1`).

Отметим использование поля `ret` в реакторе SLS: вызывается ретернер `mongo`, сообщающий системе Salt о необходимости перенаправления результатов выполнения состояния в базу данных MongoDB. Эта инструкция не является обязательной, ее можно не включать в SLS-файл.

Предположим, что у нас есть pillar-файлы, хранимые и отслеживаемые в системе Git. При необходимости конфигурирования локального клона для наблюдения за удаленным исходным сервером предыдущий пример представляет собой превосходный вариант оркестровки (orchestration): запрос на объединение (слияние) версий запускает процедуру автоматического развертывания конфигурации пиринговых узлов NTP во всей сети без каких-либо операций, выполняемых вручную. Также отметим различие между процессом управления конфигурации и процессом автоматизации, управляемой событиями: настройка маяка, реактора, SLS – всего лишь 15 строк в общей сложности, и результаты передаются в сервис структурированной базы данных. Более того, обрабатываются объекты данных, абсолютно независимые от конкретных производителей оборудования, а не файлы с каким-то предопределенным форматом.

Добавление бизнес-логики с использованием Thorium

Для реактора существует ограничение: он предоставляет возможность активизации действия только для одного события. Решить эту проблему позволяет Thorium – сложная, многокомпонентная система, дающая возможность определять бизнес-логику на основе объединенных данных и многочисленных событий.

Tlgm: @it_boooks

Как и любая другая Salt-система, Thorium имеет собственные корневые каталоги для хранения файлов, по умолчанию – */srv/thorium*, но их можно изменить в главном файле конфигурации:

```
thorium_roots:
  base:
    - /etc/salt/thorium
```

В корневом каталоге Thorium разместим требуемый файл *top.sls*, в который включены дескрипторы Thorium.

Например, предположим, что нужно передать оповещение HipChat после трех выполнений коммита (то есть трех выполнений функции `net.commit`).

Поскольку для выполнения этой задачи требуется счетчик, необходимо определить регистр, связанный с функцией `net.commit`, который будет вести счет:

```
commits:
  reg.list:
    - add: fun
    - match: salt/job/*/new
  check.contains:
    - value:
        fun: net.commit
    - count_gte: 3
```

В регистре commits сохраняются имена функций, извлеченные из событий, соответствующих шаблону тега `salt/job/*/new`. Далее `check.contains` гарантирует, что этот регистр содержит имя функции `net.commit` как минимум в трех экземплярах.

Вторая часть определяет передачу оповещения HipChat:

```
too_many_commits:
  runner.cmd:
    - fun: salt.cmd
    - arg:
      - hipchat.send_message
    - kwargs:
      - room_id: 1717
      - message: too many commits
      - from_name: ''
      - api_key: Ag56uXGGB6jTh1Lc8sEpZOgX6rMCm7M5wN6dPLFd
      - api_version: v2
    - require:
      - check: commits
```

Такая конфигурация сообщает системе Thorium о необходимости выполнения курьера `salt.cmd` для вызова выполняемой функции `hipchat.send_message` со всеми необходимыми подробностями, но только в том случае, когда обнаружено полное соответствие регистру `commits`, определенному ранее.

Теперь сохраним приведенную выше конфигурацию в SLS-файле, например *too_many_commits.sls*, в каталоге */etc/salt/thorium* и включим его в соответствующий top-файл (*/etc/salt/thorium/top.sls*):

Tlgm: @it_boooks


```
base:
  '*':
    - too_many_commits
```

После трех коммитов комплексный реактор Thorium отправит оповещение HipChat.

Это самый простой пример, но нет никаких ограничений по сложности описаний регистров и действий.



Даже без использования механизма автоматизации, управляемой событиями, с помощью системы Salt можно сделать достаточно много. Мы рекомендуем начать с применения команды `salt` и создания соответствующих SLS-файлов данных в форме pillar-файлов и шаблонов. После овладения основами практической работы с системой Salt вы будете готовы к использованию функциональных возможностей механизма автоматизации, управляемой событиями, и достигнете гораздо лучшего понимания того, что предлагает этот механизм.

Дополнительная информация о Salt

Сообщество разработало документ, содержащий наиболее эффективные практические методики и приемы обеспечения безопасности среды Salt. Этот документ включен в комплект официальной документации (https://docs.saltstack.com/en/latest/topics/hardening.html).

Мы рассмотрим одну из таких методик – ограничение для пользователей, не позволяющее им выполнять любую команду по собственному усмотрению. Используя систему Publisher ACL, можно определить права доступа (права на выполнение конкретных команд) для каждого пользователя:

```
publisher_acl:
  mircea:
    - .*
  jason:
    - csr*:
      - ntp.*
      - test.*
```

В этом примере пользователю `mircea` разрешено выполнять все команды, тогда как пользователь `jason` получает разрешение только на использование выполняемых функций из модулей `ntp` и `test`, но в применении только для миньонов `csr*`.

Чтобы ограничить доступ через REST API, можно сконфигурировать внешнюю систему аутентификации External Authentication (eAuth):

```
external_auth:
  pam:
    matt:
      - 'vmx*':
        - ntp.*
        - napalm_yang.*
    scott:
      - '@runner'
```

Tlgm: @it_boooks

Структура этой конфигурации очень похожа на конфигурацию системы Publisher ACL, в которой применяется тип аутентификации PAM (Pluggable Authentication Modules): `matt` может выполнять функции из модулей `ntp` и `napalm_yang`, а `scott` может вызвать любой курьер (runner). Отметим, что PAM – это только лишь один из вариантов использования системы аутентификации.

Механизм малых запросов к базам данных (SDB)

Интерфейс SDB (small database queries) предназначен для хранения и извлечения данных, но, в отличие от интерфейсов pillars и grains, не обязательно связанных с миньонами. Работа с данными осуществляется через малые запросы к базе данных (отсюда имя SDB) с использованием компактного URI. Это дает возможность пользователям быстро обращаться к значениям баз данных из различных областей конфигурации Salt без существенных издержек. Основной формат SDB URI: *sdb://<proile>/<args>*.

Доступно несколько SDB-модулей, в том числе: SQLite3, CouchDB, Consul, Keyring, Memcached, вызовы REST API, Vault, а также переменные среды. В частности, для Vault конфигурация весьма проста, как показано в следующем примере.

Файл */etc/salt/minion* или */etc/salt/master*:

```
myvault:
  driver: vault
```

После конфигурирования в миньоне можно получить доступ к данным из любого файла конфигурации (*/etc/salt/master*, */etc/salt/minion* или */etc/salt/proxy*), используя URI следующим образом:

```
password: sdb://myvault/secret/passwords?get_pass
```

Здесь *myvault* – профиль конфигурации, *secret/passwords* – путь, по которому осуществляется доступ к данным, а *get_pass* – ключ для возвращаемых данных.

Интерфейс SDB предоставляет удобный способ управления часто изменяющимися данными и позволяет избежать повторения одной и той же конфигурации в нескольких местах, предоставляя для использования простой URI.

Кеш системы Salt

Salt кеширует разнообразные данные, в основном способствующие улучшению производительности, но этим не ограничивается. Одним из наиболее важных является кеширование рабочего задания – после выполнения задания результаты сохраняются в течение 24 часов – по умолчанию в локальной файловой системе */var/cache/salt/master/jobs*. Можно отключить кеширование рабочих заданий с помощью параметра `job_cache: false` или настроить интервал кеширования в параметре `keep_jobs` (в часах) в главном файле конфигурации. Кроме того, есть возможность перенаправлять любую информацию в независимый сервис. Можно повторно использовать конфигурацию ретернеров, описанную выше, и получить множество вариантов выбора: Cassandra, Elasticsearch, MySQL, Redis, Slack, SMS, SMTP и т. д. Например, при необходимости

Tlgm: @it_boooks

можно даже организовать наблюдение за всей деятельностью системы Salt из комнаты HipChat.

Grains- и pillar-данные также кешируются на сервере-координаторе. По умолчанию выбирается локальная файловая система, но место хранения данных можно выбрать и на удаленной системе или воспользоваться локальной базой данных Redis, размещенной в оперативной памяти.

SLS-файлы, определенные в путях `file_roots` или извлеченные из внешних сервисов, также кешируются, но обновляются при любом изменении их содержимого в исходных локациях. Такая оптимизация чрезвычайно важна, когда (прокси-) миньон и сервер-координатор работает на отдельных физических машинах (между которыми могут устанавливаться ненадежные или медленные соединения), поскольку файлы не передаются, пока в этом нет необходимости. При использовании больших файлов это важное улучшение в плане скорости выполнения. По умолчанию принят путь к кеш-файлам */var/cache/salt/minion* для обычных миньонов и путь */var/cache/salt/proxy/<ID>* для прокси-миньонов. Эти пути также можно изменить с помощью параметра `cachedir`.

Ведение журналов в системе Salt

Система Salt перехватывает и фиксирует в журналах всевозможную информацию из приложений, работающих под управлением и наблюдением Salt. Уровень подробности ведения журналов настраивается с помощью параметра `log_level` в главном файле конфигурации или для отдельных миньонов (каждое приложение имеет собственный, независимый от других процесс журналирования). Возможны следующие уровни: `garbage`, `trace`, `debug`, `info`, `warning`, `error` или `critical`, но следует помнить, что при установке первых трех уровней в журналы могут вноситься конфиденциальные данные.

Для более детальной организации ведения журналов можно устанавливать различные уровни для каждого типа модуля:

```
log_granular_levels:
  salt.states: warning
  salt.beacon: error
  salt.modules: info
```

Расширения системы Salt

В текущем разделе постоянно подчеркивается, что каждый компонент системы Salt является подключаемым. Модули расширения могут быть размещены в каталоге с подкаталогами, предназначенными для каждого типа модулей, то есть *modules*, *states*, *returners*, *output* и *runners*. Соглашение об именовании подкаталогов предусматривает наличие символа подчеркивания, предшествующего имени типа модуля, таким образом, выполняемые модули определяются в подкаталоге *_module*, курьеры – в подкаталоге *_runners*, модули вывода – в подкаталоге *_output*. Родительский каталог может быть задан с помощью параметра `extension_modules` или `module_dirs`, в котором указывается список путей. Кроме того, список путей к модулям расширения можно включить в один из путей `file_roots`.

Tlgm: @it_boooks


```
extension_modules
  - /etc/salt/extmods
```

Например, новый модуль расширения с именем *example.py* можно поместить в подкаталог */etc/salt/extmods/_modules*:

```
# -*- coding: utf-8 -*-
def test():
    return {
      'network_programming_with_salt': True
    }
```

Чтобы система Salt узнала об этом новом модуле, необходимо ресинхронизировать все модули с помощью выполняемой функции `saltutil.sync_all`:

```
$ sudo salt vmx1 saltutil.sync_all
vmx1:
    ----------
    beacons:
    clouds:
    engines:
    grains:
    log_handlers:
    modules:
        - modules.example
    output:
    proxymodules:
    renderers:
    returners:
    sdb:
    states:
    utils:
```

Здесь мы видим объект `modules.example`, подтверждающий, что новый выполняемый модуль `example` синхронизирован и доступен для вызова:

```
$ sudo salt vmx1 example.test
vmx1:
    ----------
    network_programming_with_salt:
        True
```

Напомним, что модули могут использоваться из командной строки или непосредственно в SLS-файлах как шаблоны:

```
{%- set successful = salt.example.test()['network_programming_with_salt'] -%}
```

Краткий итоговый обзор системы Salt

В этом разделе мы рассмотрели некоторые наиболее важные темы, позволяющие начать процесс автоматизации сети с использованием системы Salt. Одним из самых значимых компонентов системы Salt являются SLS-файлы.

Tlgm: @it_boooks

Вы узнали, как формируются SLS-файлы данных с использованием языков Jinja и YAML (по умолчанию), с применением Mako и HJSON, а также с добавлением нового или специализированного языка шаблонов или даже формата данных. Это позволяет гибко использовать Salt и расширять возможности системы в соответствии с требованиями эксплуатационной среды независимо от официальной кодовой базы. Другим важным преимуществом Salt является использование прокси-миньонов. В Salt предлагается встроенная возможность распределения нагрузки между прокси-миньонами. Благодаря этому функциональному механизму Salt является едва ли не наилучшим вариантом выбора для работы с крупномасштабными и распределенными сетями.

Автоматизация сети, управляемая событиями, с использованием StackStorm

StackStorm – это программный проект с открытым исходным кодом, предназначенный для поддержки гибкого процесса автоматизации, управляемого событиями. Этот проект часто ассоциируют с другими инструментальными средствами, рассматриваемыми в текущей главе, но в действительности StackStorm изначально не предназначался для замены существующих инструментов управления конфигурацией. Например, многие наиболее часто используемые рабочие процессы (потоки) в StackStorm в действительности применяют инструменты, подобные Ansible, для выполнения задач управления конфигурацией.

Чтобы наилучшим образом усвоить принципы работы StackStorm, необходимо понять, что он полностью соответствует наиболее выгодной позиции между процессом управления конфигурацией (или процессом общей автоматизации) и процессом мониторинга. StackStorm ориентирован на предоставление набора простейших программных компонентов, позволяющих пользователю описывать задачи, которые должны выполняться в ответ на определенные события. Поэтому StackStorm можно считать инструментальным средством класса IFTTT (if-this-then-that – если произошло что-то, то сделать это) для ИТ-инфраструктуры.

Во многих областях ИТ ответная реакция на проблемы или перебои в работе обычно осуществляется вручную. Одним из наиболее часто применяемых вариантов использования StackStorm является концепция автокоррекции, то есть принцип попыток решения проблем без вмешательства человека. Но это, разумеется, не волшебная кнопка, нажав которую, можно автоматически устранить все проблемы. Тем не менее автокоррекция является реализацией идеи, согласно которой после решения проблемы вручную необходимо зафиксировать эту процедуру как своеобразный автоматизированный рабочий процесс (поток), со временем сокращая количество задач, выполняемых вручную. Именно эту задачу позволяет решить StackStorm.

Tlgm: @it_boooks

Проектное решение StackStorm предполагает интенсивное применение практических методик типа IaC (Infrastructure as Code). В этих методиках считается, что инфраструктура может быть описана с использованием текстовых файлов (например, файлов на языке YAML и шаблонов Jinja, которые многократно встречались вам в данной книге). Таким образом, эти файлы могут и должны управляться тем же способом, которым разработчики ПО управляют исходным кодом создаваемых приложений.

В системе StackStorm почти все описывается с помощью файлов на языке YAML. Помните об этом принципе при рассмотрении следующих примеров. Также напомним, что удачным решением всегда является использование системы управления версиями и автоматического тестирования файлов, предназначенных для управления инфраструктурой.

В следующем разделе будут рассматриваться некоторые основные концепции StackStorm, которые необходимо хорошо понимать, чтобы начать работу с этой системой. Вы увидите, что многие внутренние концепции StackStorm основаны на методическом подходе «инфраструктура как код» (Infrastructure as Code – IaC) – в системе StackStorm все определяется через обычные текстовые файлы. Поэтому вы можете (и даже должны) управлять всеми объектами в StackStorm, используя приемы и принципы, изученные в главе 8.

Основные концепции системы StackStorm

Существует ряд концепций (основных принципов), которые необходимо хорошо знать, чтобы использовать систему StackStorm (см. рис. 9.4). В совокупности все эти концепции образуют набор инструментальных средств, предоставляющий возможности реализации процесса автоматизации, управляемого событиями.

Операции (*actions*) по своей сущности ближе всего к функциональным возможностям, предлагаемым некоторыми другими инструментальными средствами, рассматриваемыми в текущей главе. Операции выполняют реальную работу, но логически немного отличаются от программного кода, который реализует такие задачи, как вызовы методов API и выполнение скриптов. Операции – это атомарные конструктивные блоки, относящиеся именно к области «автоматизации» в общем рабочем процессе автоматизации, управляемой событиями.

Далее необходимо выделить *рабочие потоки*, или *процессы* (*workflows*). Рабочие потоки – это способ связывания операций в единое целое для реализации требуемой бизнес-логики. Рабочий поток может начаться с выполнения нескольких операций для сбора некоторых дополнительных данных из управляемой инфраструктуры, затем эти данные используются для принятия решений о выборе следующих операций, предназначенных для устранения проблемы. В системе StackStorm рабочие потоки доступны в двух формах. Простейший формат – ActionChains, представляющий элементарный способ объединения операций в «цепочку». Второй формат – Mistral, один из проектов инициативы OpenStack с собственным способом (форматом) определения рабочего потока. Этот проект входит в основной комплект StackStorm и предоставляет гораздо более гибкие и универсальные возможности.

Tlgm: @it_boooks

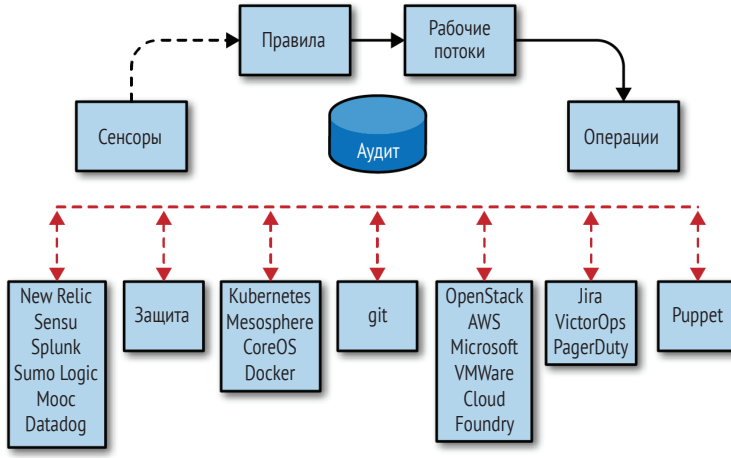


Рис. 9.4 ❖ Концепции StackStorm

Предназначение и основные функциональные возможности операций и рабочих потоков сами по себе понятны, но теперь необходимо перенести эти концепции в среду автоматизации, управляемой событиями. Для этого используются сенсоры и триггеры. *Сенсоры* (*sensors*) можно считать небольшими фрагментами кода на языке Python, единственной задачей которых является сбор данных об управляемой инфраструктуре. Отметим, что это не агенты, которые развертываются в конечных пунктах, как серверы или сетевые устройства, в отличие от них, сенсоры работают внутри самой системы StackStorm и обычно программно устанавливают соединения с внешними объектами, такими как системы мониторинга или системы управления, или в некоторых случаях с дискретными узлами, такими как виртуальные машины или сетевые устройства, чтобы собирать требуемую информацию. Но эти данные не имеют смысла, если отсутствует способ распознавания конкретного события, связанного с их появлением. Поэтому сенсоры также могут определять *триггеры* (*triggers*), которые срабатывают при возникновении сколько-нибудь значимого события, то есть фактически определяют «источник» данных, получаемых сенсором. Например, триггер `napalm.LLDPNeighborDecrease` оповещает StackStorm, когда заданное сетевое устройство обнаруживает сокращение списка соседних узлов LLDP.

Сущность автоматизации, управляемой событиями, в StackStorm реализована в форме правил. *Правила* (*rules*) системы StackStorm фактически равнозначны принципу «если произошло что-то, то выполнить это» (if-this-then-that), то есть предоставляют способ установления соединения между входящими триггерами и операциями или рабочими потоками, которые отвечают на события. Может потребоваться выполнение рабочего потока, который продвигает новую конфигурацию на сетевое устройство при срабатывании триггера

Tlgm: @it_boooks

napalm.LLDPNeighborDecrease. Такая автоматическая ответная реакция должна быть определена в соотвествующем правиле.

Наконец, заслуживает упоминания и тот факт, что все описанные выше концепции распространяются в системе StackStorm с помощью пакетов. Пакеты (packs) – это элементарные неделимые единицы или блоки распространения для всех способов и путей расширения системы StackStorm – сенсоров, правил, рабочих потоков и операций, которые все определены в текстовых файлах. Сами файлы размещены в пакете, на который можно ссылаться, а также обращаться к его содержимому. Например, сенсор napalm.LLDPNeighborDecrease размещен в пакете napalm. Не все пакеты устанавливаются по умолчанию, но существуют команды для простой установки новых пакетов из репозитория StackStorm Exchange. Например, для загрузки и установки пакета napalm нужно выполнить следующую команду:

```
st2 pack install napalm
```

После этого все операции, рабочие потоки, сенсоры и правила, содержащиеся в пакете napalm, становятся доступными в вашем экземпляре StackStorm.

Теперь основные концепции понятны, и мы переходим к изучению внутренней структуры StackStorm.

Архитектура StackStorm

Система StackStorm представляет собой не один монолитный компонент, это группа микросервисов, работающих совместно для реализации возможностей автоматизации, управляемой событиями (см. рис. 9.5). Распределенная сущность системы была спроектирована так, чтобы для каждого компонента существовала возможность независимого масштабирования в соответствии с текущими требованиями и необходимыми изменениями. Кроме того, такой подход обеспечивает более высокую жизнеспособность каждой отдельной функции при фатальных системных сбоях и аварийных ситуациях.

Например, если известно, что потребуется большая мощность для рабочих потоков, но не будет необходимости наблюдать за множеством событий, чтобы инициировать и завершать соответствующие действия, то можно воспользоваться некоторыми дополнительными компонентами st2actionrunner для регулирования рабочей нагрузки. Если необходимо обрабатывать большое количество событий, то рекомендуется расширить эти возможности с помощью компонента st2sensorcontainer аналогичным образом.

Отметим, что ни один из названных компонентов не является «агентом», то есть вам не придется отдельно устанавливать эти компоненты в других частях инфраструктуры. Все, что показано на рис. 9.5, остается внутри системы StackStorm, работающей на серверах или сетевых узлах.

Один из компонентов заслуживает особого внимания – st2web. Так кратко и удобно обозначено имя пользовательского веб-интерфейса, входящего в дистрибутивный комплект StackStorm (см. рис. 9.6).

Tlgm: @it_boooks

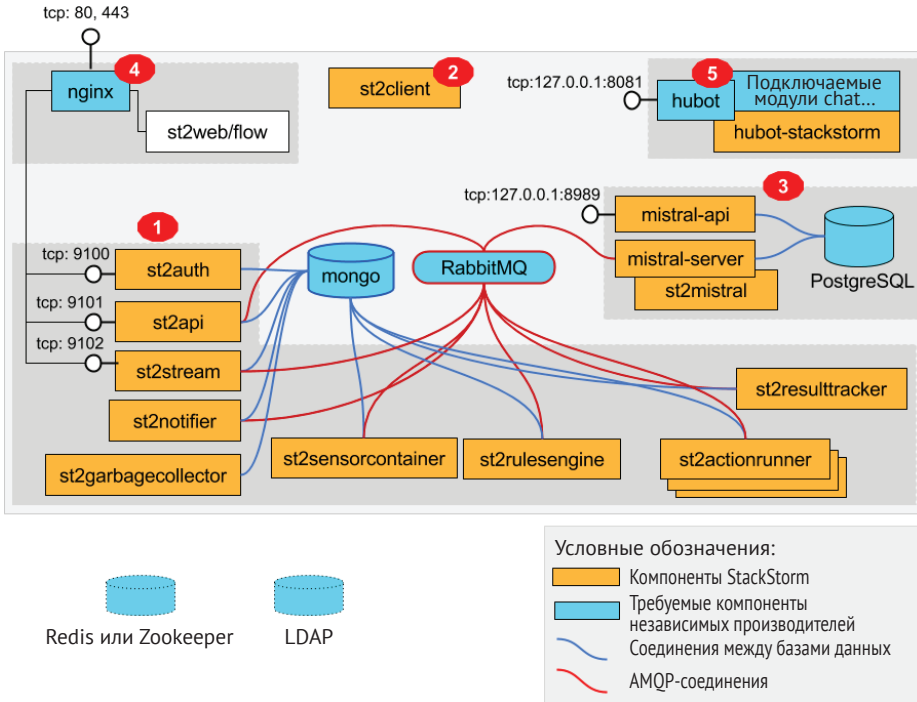


Рис. 9.5 ❖ Компоненты системы StackStorm

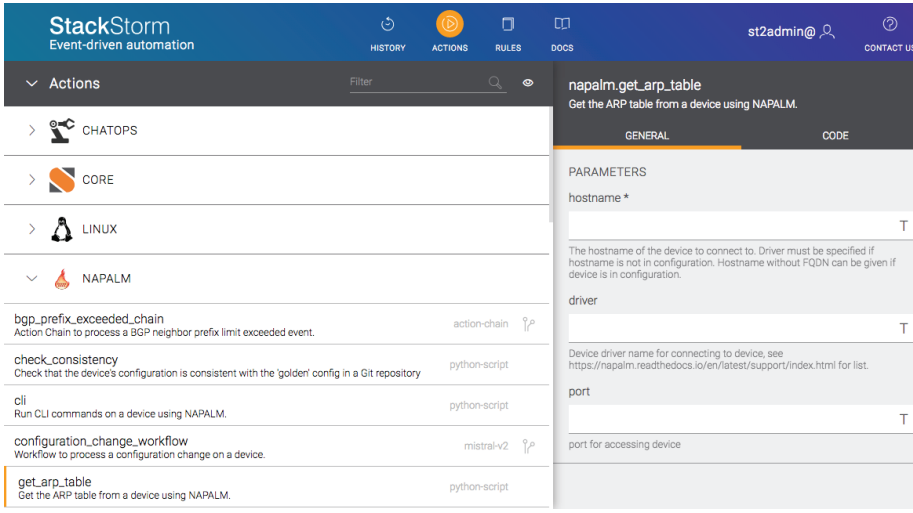


Рис. 9.6 ❖ Пользовательский веб-интерфейс в системе StackStorm

Tlgm: @it_boooks

С помощью этого веб-интерфейса в системе StackStorm можно делать практически все. Даже если вы твердо решили работать в командной оболочке bash, то все же следует помнить о том, что существует более удобный и дружественный к пользователю вариант. В следующих примерах будет использоваться интерфейс командной строки StackStorm, поскольку он более нагляден и понятен при обучении.

Операции и рабочие потоки

После освоения основных концепций и понимания архитектуры системы на высоком уровне необходимо перейти к практическим примерам, демонстрирующим все, что было теоретически изложено до настоящего момента.



StackStorm – чрезвычайно мощная система, именно поэтому она требует подробнейшего изучения и обсуждения. Но мы не будем заполнять этот раздел огромным количеством примеров, показывающих абсолютно все, что вы, вероятно, хотите знать, а ограничимся обзором наиболее важных подробностей. Всю прочую информацию можно узнать из документации StackStorm (https://docs.stackstorm.com), и мы рекомендуем как можно чаще обращаться к ней, чтобы получить в свое распоряжение гораздо более подробные описания, фрагменты кода и многое другое.

Кроме того, существует весьма активное сообщество Slack (к нему можно присоединиться на сайте https://stackstorm.com/?#community). Здесь всегда можно задать интересующие вас вопросы и получить качественные ответы.

Вы можете самостоятельно двигаться дальше – для этого рекомендуется посетить репозиторий st2vagrant (https://github.com/StackStorm/st2vagrant). Там вы найдете файл Vagrantfile и несколько скриптов для упрощенного развертывания одного экземпляра системы StackStorm.



Инструментальное средство Vagrant подробно рассматривается в главе 10.

Начнем с самого простого – с выполнения одной команды `echo`, выводящей классическую фразу «Hello World». Для этого воспользуемся операцией `core.local`, позволяющей выполнить любую команду, которая возможна в командной оболочке bash.



Разнообразие внутренних скрытых механизмов можно использовать в операциях для выполнения реальной работы. Например, `core.local` просто передает заданное значение в командную оболочку bash, но операция может использовать язык Python или скрипт командной оболочки для управления более сложной логикой.

В комплект StackStorm включен собственный интерфейс командной строки: команда `st2`. Одной из доступных подкоманд является `st2 run`, позволяющая напрямую выполнить действие или рабочий поток, не обращаясь к сенсорам или правилам.

Можно выполнить команду `st2 run core.local -h`, чтобы узнать, какие параметры требуются для этой операции:

Tlgm: @it_boooks


```
vagrant@st2vagrant:~$ st2 run core.local echo -h

Action that executes an arbitrary Linux command on the localhost.
(Операция, выполняющая любую команду Linux на локальном хосте.)

Required Parameters:
(Обязательные параметры:)
    cmd
        Arbitrary Linux command to be executed on the local host.
        (Любая команда Linux, которую необходимо выполнить на локальном хосте.)
        Type (тип): string

Optional Parameters:
(Необязательные параметры:)
    cwd
        Working directory where the command will be executed in
        (Рабочий каталог, в котором будет выполнена команда)
        Type (тип): string

    env
        Environment variables which will be available to the command(e.g.
        key1=val1,key2=val2)
        (Переменные среды, доступные для выполняемой команды (например,
key1=val1,key2=val2))
        Type (тип): object

    kwarg_op
        Operator to use in front of keyword args i.e. "--" or "-".
        (Оператор, используемый перед ключевым словом args, то есть "--" или "-".)
        Type (тип): string
        Default (по умолчанию): --

    timeout
        Action timeout in seconds. Action will get killed if it doesn't finish
        in timeout seconds.
        (Тайм-аут для операции в секундах. Операция будет принудительно удалена (уничтожена),
         если не завершит работу за интервал тайм-аута, заданный в секундах.)
        Type (тип): integer
        Default (по умолчанию): 60
```

Как показано в предыдущем примере, для операции `core.local` обязательным является один позиционный параметр – команда, которую должна выполнить эта операция (в нашем случае это команда `echo`):

```
vagrant@st2vagrant:~$ st2 run core.local echo "Hello World!"
.
id: 59598d2bc4da5f0506c24981
status: succeeded
parameters:
  cmd: echo Hello World!
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: Hello World!
  succeeded: true
```

Tlgm: @it_boooks

Вывод показывает, что команда выполнена успешно и поток вывода `stdout` содержит строку, переданную в команду `echo`.

К этому моменту мы уже вполне уверенно чувствуем себя в командной строке, поэтому можем рассмотреть пример, несколько более приближенный к задачам автоматизации сети. Предположим, что предварительно уже был установлен пакет `napalm` с помощью команды `st2 pack install napalm`, поэтому можно воспользоваться командой `st2 actions list` для просмотра всех доступных операций в этом пакете:

```
vagrant@st2vagrant:~$ st2 action list --pack=napalm
+-------------------------------------+
| ref                                 |
+-------------------------------------+
| napalm.bgp_prefix_exceeded_chain    |
| napalm.check_consistency            |
| napalm.cli                          |
| napalm.configuration_change_workflow |
| napalm.get_arp_table                |
| napalm.get_bgp_config               |
| napalm.get_bgp_neighbors            |
| napalm.get_bgp_neighbors_detail     |
| napalm.get_config                   |
| napalm.get_environment              |
| napalm.get_facts                    |
| napalm.get_firewall_policies        |
| napalm.get_interfaces               |
| napalm.get_lldp_neighbors           |
| napalm.get_log                      |
| napalm.get_mac_address_table        |
| napalm.get_network_instances        |
| napalm.get_ntp                      |
| napalm.get_optics                   |
| napalm.get_probes_config            |
| napalm.get_probes_results           |
| napalm.get_route_to                 |
| napalm.get_snmp_information         |
| napalm.interface_down_workflow      |
| napalm.loadconfig                   |
| napalm.ping                         |
| napalm.traceroute                   |
+-------------------------------------+
```

Для практического использования этого пакета необходимо сконфигурировать его так, чтобы он понимал, как можно получить доступ к нашим сетевым устройствам и как необходимо выполнять процедуры аутентификации на этих устройствах. Конфигурация всех пакетов выполняется в YAML-файлах, расположенных в каталоге */opt/stackstorm/configs*, следовательно, в нашем случае это будет файл */opt/stackstorm/configs/napalm.yaml*. Минимальная конфигурация показана ниже:

Tlgm: @it_boooks


```
---
html_table_class: napalm
config_repo: https://github.com/StackStorm/vsrx-configs.git

credentials:
  local:
    username: root
    password: Juniper

devices:
- hostname: vsrx01
  driver: junos
  credentials: local
```

В этой конфигурации определено одно устройство с именем хоста `vsrx01`, используются регистрационные данные `local`, то есть имя пользователя `root` и пароль `Juniper`.

При внесении изменений в конфигурацию важно, чтобы система StackStorm получила информацию об этих изменениях, поэтому измененную конфигурацию необходимо перезагрузить. Для этой цели весьма полезна утилита `st2ctl`:

```
vagrant@st2vagrant:~$ st2ctl reload --register-configs
Registering content...[flags = --config-file /etc/st2/st2.conf --register-configs
2017-07-03 01:49:44,941 INFO [-] Connecting to database "st2" @ "0.0.0.0:27017" a
user "stackstorm".
2017-07-03 01:49:45,056 INFO [-] ================================================
2017-07-03 01:49:45,056 INFO [-] ############## Registering configs #############
2017-07-03 01:49:45,056 INFO [-] ================================================
2017-07-03 01:49:45,181 INFO [-] Registered 1 configs.
##### st2 components status #####
st2actionrunner PID: 1007
st2actionrunner PID: 1053
st2api PID: 891
st2api PID: 1286
st2stream PID: 893
st2stream PID: 1288
st2auth PID: 874
st2auth PID: 1287
st2garbagecollector PID: 872
st2notifier PID: 884
st2resultstracker PID: 879
st2rulesengine PID: 888
st2sensorcontainer PID: 864
st2chatops PID: 881
mistral-server PID: 1032
mistral-api PID: 987
mistral-api PID: 2703
mistral-api PID: 2706
```



Во время написания данной книги все операции в пакете `napalm` обязательно требовали наличия как минимум одного аргумента, а именно `hostname`. Это позволяет операции точно знать, с каким устройством в текущей конфигурации пакета вы намерены работать.

Tlgm: @it_boooks

Теперь мы полностью готовы к выполнению операций NAPALM. Операция `napalm.get_facts` извлекает факты о заданном сетевом устройстве:

```
vagrant@st2vagrant:~$ st2 run napalm.get_facts hostname=vsrx01
..
id: 5959a495c4da5f0506c2498a
status: succeeded
parameters:
  hostname: vsrx01
result:
  exit_code: 0
  result:
    raw:
      fqdn: vsrx01
      hostname: vsrx01
      interface_list:
      - ge-0/0/0
      - ge-0/0/1
      - ge-0/0/2
      - ge-0/0/3
      model: FIREFLY-PERIMETER
      os_version: 12.1X47-D15.4
      serial_number: da12e84e2e72
      uptime: 240
      vendor: Juniper
  stderr: ''
  stdout: ''
```

Здесь можно видеть некоторую полезную информацию о сетевом устройстве, например имя производителя и список существующих интерфейсов.

Операции по своей сущности действительно предназначены для правильного выполнения одной задачи. Но в реальной эксплуатационной среде повседневно выполняемые задачи сетевой инфраструктуры редко принимают форму одной задачи. Обычно работа включает несколько отдельных задач и блоки принятия решений в процессе выполнения.

Например, если получено сообщение о том, что маршрутизатор перешел в режим онлайн, может потребоваться сбор дополнительной информации от работающих вместе с ним узлов. Возможно, необходимо проверить работоспособность кабелей. При возникновении других проблем, вероятнее всего, потребуется совершенно другой набор задач. Как уже было отмечено выше, рабочие потоки позволяют использовать операции более изощренными способами, обеспечивая внесение всех этих сложных решений в текстовый файл, как если бы это был исходный программный код.

Соблюдая краткости изложения, мы рассмотрим только один из вариантов рабочих потоков в системе StackStorm – Mistral. Mistral – это проект инициативы OpenStack, который предоставляет следующие две функциональные возможности:

- стандартизированный язык на основе YAML для определения рабочих потоков;

Tlgm: @it_boooks

- программное обеспечение с открытым исходным кодом для приема и обработки запросов на выполнение в рабочих потоках.



Если вы устанавливаете StackStorm, пользуясь инструкциями официальной документации, то Mistral устанавливается и запускается вместе с другими процессами StackStorm.

Рассмотрим простой пример рабочего потока Mistral, который позволит вам более подробно узнать, как он работает.

```
---
version: '2.0'

examples.mistral-basic:
    description: A basic workflow that runs an arbitrary linux command.
    type: direct
    input:
        - cmd
    output:
        stdout: "{{ _.cmd }}"
    tasks:
        task1:
            action: core.local cmd="{{ _.cmd }}"
            publish:
                stdout: "{{ task('task1').result.stdout }}"
```

У этого рабочего потока есть несколько важных основных характеристик:

- `input` – здесь объявляются параметры для рабочего потока. Они публикуются в рабочем потоке как именованные переменные, которые можно использовать в других задачах;
- `output` – управляет выводом, то есть определяет, значения каких переменных из рабочего потока необходимо опубликовать после его завершения;
- `tasks` – содержит список задач. В приведенном выше простом примере список включает только одну задачу: `task1`. Отметим, что эта задача не только ссылается на операцию, которую необходимо выполнить (здесь: `core.local`), но также содержит ключ `publish`, который передает значения из потоков `stdout` и `stderr` в переменные с аналогичными именами (обратите внимание на то, что для этого используются небольшие фрагменты на языке Jinja). Также отметим, что поток вывода `stdout` должен передаваться в переменную `output`, чтобы можно было видеть полученный результат после завершения рабочего потока.

Рабочие потоки выполняются тем же способом, который был описан выше для выполнения операций. Ранее уже отмечалось, что реальная работа, выполняемая операциями, может принимать форму скрипта на языке командной оболочки bash или Python, может быть единственной простой командой или, как в нашем случае, представлять собой рабочий поток Mistral. Здесь можно еще раз воспользоваться командой `st2 run`, чтобы обеспечить передачу обязательного параметра `cmd`, который рабочий поток, в свою очередь, передает в уже знакомую нам операцию `core.local`:

Tlgm: @it_boooks


```
vagrant@st2vagrant:~$ st2 run examples.mistral-basic cmd="echo Hello, Mistral!"
.
id: 595ad9c3c4da5f0521ea906c
action.ref: examples.mistral-basic
parameters:
  cmd: echo Hello, Mistral!
status: succeeded
result_task: task1
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: Hello, Mistral!
  succeeded: true
start_timestamp: 2017-07-03T23:56:51.069066Z
end_timestamp: 2017-07-03T23:56:52.442155Z
+--------------------------+----------------------+-------+------------+
| id                       | status               | task  | action     |
+--------------------------+----------------------+-------+------------+
| 595ad9c3c4da5f0521ea906f | succeeded (0s elapsed) | task1 | core.local |
+--------------------------+----------------------+-------+------------+
```

В этом случае вывод немного отличается от предыдущего примера. При каждом запуске операции система StackStorm создает операцию «execution». Операция execution представляет всю информацию о том, как выполняется заданная операция. Отметим, что в последнем примере, как и в предыдущих, при каждом вызове `st2 run` создается идентификатор выполняемой операции (execution ID). В примере для рабочего потока Mistral идентификатор указан в первой строке вывода, но, кроме него, можно видеть еще и таблицу выполняемых потомков, которые являются задачами в данном рабочем потоке.

В качестве примера, в большей степени ориентированного на сетевую среду, рассмотрим один из рабочих потоков Mistral в пакете `napalm` (код отредактирован в целях экономии места):

```
---
version: '2.0'

napalm.interface_down_workflow:

  input:
    - hostname
    - interface

  type: direct

  tasks:

    show_interface:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
      on-success: "show_interface_counters"
```

Tlgm: @it_boooks


```
    show_interface_counters:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
        counters: true
      on-success: "show_log"

    show_log:
      action: "napalm.get_log"
      input:
        hostname: "{{ _.hostname }}"
        lastlines: 10
```

Обратите внимание на две первые задачи, в которых используется ключевое слово `on-success`, для того чтобы определить, какая задача должна выполняться следующей (указанная задача будет выполнена, если включающая ее предыдущая задача завершилась успешно (status: success)).

При выполнении этого рабочего потока получим результат в уже знакомой форме, показывающий три выполняемые операции – по одной для каждой задачи, определенной в этом рабочем потоке:

```
vagrant@st2vagrant:~$ st2 run napalm.interface_down_workflow hostname=vsrx01
interface="ge-0/0/1"
......
id: 595adf58c4da5f0521ea90a0
action.ref: napalm.interface_down_workflow
parameters:
  hostname: vsrx01
  interface: ge-0/0/1
status: succeeded
start_timestamp: 2017-07-04T00:20:40.895706Z
end_timestamp: 2017-07-04T00:20:52.735536Z
+--------------------------+-----------+------------------------+----------------------+
| id                       | status    | task                   | action               |
+--------------------------+-----------+------------------------+----------------------+
| 595adf59c4da5f0521ea90a3 | succeeded | show_interface         | napalm.get_interfaces |
| 595adf5cc4da5f0521ea90a5 | succeeded | show_interface_counters | napalm.get_interfaces |
| 595adf5fc4da5f0521ea90a7 | succeeded | show_log               | napalm.get_log        |
+--------------------------+-----------+------------------------+----------------------+
```

Теперь можно воспользоваться командой `st2 execution get <id>`, чтобы увидеть подробный результат одной из выполненных операций.

```
vagrant@st2vagrant:~$ st2 execution get 595adf5cc4da5f0521ea90a5
id: 595adf5cc4da5f0521ea90a5
status: succeeded (3s elapsed)
parameters:
  counters: true
  hostname: vsrx01
  interface: ge-0/0/1
result:
```

Tlgm: @it_boooks


```
  exit_code: 0
  result:
    raw:
      name: ge-0/0/1
      rx_broadcast_packets: 0
      rx_discards: 0
      rx_errors: 0
      rx_multicast_packets: 0
      rx_octets: 0
      rx_unicast_packets: 0
      tx_broadcast_packets: 0
      tx_discards: 0
      tx_errors: 0
      tx_multicast_packets: 0
      tx_octets: 0
      tx_unicast_packets: 0
  stderr: ''
  stdout: ''
```

Кроме того, можно использовать некоторые простые логические конструкции ветвления в Mistral для принятия несколько более сложных решений в рабочем потоке. Следующий пример является слегка измененной версией предыдущего, с новой задачей, добавленной в начальную часть:

```
---
version: '2.0'

napalm.interface_down_workflow:
  input:
    - hostname
    - interface
    - skip_show_interface

  type: direct

  tasks:

    decide_task:
      action: "core.noop"
      on-success:
      - show_interface: "{{ _.skip_show_interface != True }}"
      - show_interface_counters: "{{ _.skip_show_interface == True }}"

    show_interface:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
      on-success: "show_interface_counters"

    show_interface_counters:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
```

Tlgm: @it_boooks


```
        interface: "{{ _.interface }}"
        counters: true
      on-success: "show_log"

    show_log:
      action: "napalm.get_log"
      input:
        hostname: "{{ _.hostname }}"
        lastlines: 10
```

Операция `core.noop` не выполняет абсолютно никаких действий. Это общепринятый способ принятия предварительных решений (на начальной стадии) в рабочем потоке Mistral. Значение ключа `on-success` для этой задачи представлено списком, а не простой строкой, определяющей следующую задачу. В этом случае условные выражения, записанные в каждом пункте списка, определяют следующую задачу, которую необходимо выполнить. Можно передать параметр `skip_show_interface` в рабочий поток и собственными глазами увидеть, что задача `show_interface` не будет выполнена.

```
vagrant@st2vagrant:~$ st2 run napalm.interface_down_workflow hostname=vsrx01
interface="ge-0/0/1" skip_show_interface=True
....
id: 595b4a7ec4da5f0521ea90b4
action.ref: napalm.interface_down_workflow
parameters:
  hostname: vsrx01
  interface: ge-0/0/1
  skip_show_interface: true
status: succeeded
start_timestamp: 2017-07-04T07:57:50.606796Z
end_timestamp: 2017-07-04T07:57:58.032456Z
+--------------------------+-----------+-------------------------+----------------------+
| id                       | status    | task                    | action               |
+--------------------------+-----------+-------------------------+----------------------+
| 595b4a7ec4da5f0521ea90b7 | succeeded | decide_task             | core.noop            |
| 595b4a7fc4da5f0521ea90b9 | succeeded | show_interface_counters | napalm.get_interfaces |
| 595b4a81c4da5f0521ea90bb | succeeded | show_log                | napalm.get_log       |
+--------------------------+-----------+-------------------------+----------------------+
```

С помощью операций и рабочих потоков можно сделать гораздо больше, но рассмотренных выше примеров вполне достаточно, чтобы начать работу с ними.

Сенсоры и триггеры

Сами по себе операции и рабочие потоки весьма полезны, но для осуществления процесса автоматизации, управляемой событиями, необходимо собирать информацию об инфраструктуре и отслеживать возникновение событий, требующих ответной реакции. Эти задачи выполняются с помощью сенсоров и триггеров. *Сенсоры* (*sensors*) – это небольшие фрагменты кода на языке Python, которые передают внешние данные в систему StackStorm, например по

Tlgm: @it_boooks

результатам периодического опроса программных интерфейсов REST API или в форме подписки на очереди сообщений.



Система StackStorm также позволяет конфигурировать входящие обратные HTTP-вызовы – webhooks (https://docs.stackstorm.com/webhooks.html), которые позволяют внешним системам «проталкивать» события в StackStorm (сенсоры в большей степени поддерживают модель взаимодействия, основанную на «втягивании» внешних событий). Сенсоры являются предпочтительным методом взаимодействия, поскольку обеспечивают более детализированное и тесное объединение, но webhooks предлагают простой механизм взаимодействия, позволяющий быстро передавать данные в систему StackStorm. Это может оказаться полезным при работе с системами, в которых нет встроенных сенсоров, или с системами, которые в качестве механизма взаимодействия предлагают только исходящие webhooks.

Просмотреть список сенсоров, доступных в системе, можно с помощью команды `st2 sensor list` (воспользуемся флагом `pack`, для того чтобы ограничиться сенсорами из пакета `napalm`):

```
vagrant@st2vagrant:~$ st2 sensor list --pack=napalm
+------------------------+--------+-----------------------------------------+---------+
| ref                    | pack   | description                             | enabled |
+------------------------+--------+-----------------------------------------+---------+
| napalm.NapalmLLDPSensor | napalm | Sensor that uses NAPALM to retrieve LLDP  | True    |
|                        |        | information from network devices        |         |
+------------------------+--------+-----------------------------------------+---------+
```

Этот сенсор периодически отправляет запросы на каждое устройство, определенное в файле конфигурации для таблицы соседних устройств LLDP. Это позволяет отслеживать количество активных соседних узлов LLDP для каждого устройства.

Как было отмечено ранее, *триггеры* (*triggers*) – это способ, которым система StackStorm узнает о возникновении события. Например, если счетчик соседних узлов LLDP остается неизменным, то нет необходимости в каких-либо действиях. Но если это количество изменяется, то возникает событие, требующее некоторых ответных действий. В следующем примере рассматриваются два триггера: первый отслеживает увеличение соседних узлов, второй – уменьшение соседних узлов:

```
vagrant@st2vagrant:~$ st2 trigger list --pack=napalm
+---------------------------+--------+-------------------------------------------------+
| ref                       | pack   | description                                     |
+---------------------------+--------+-------------------------------------------------+
| napalm.LLDPNeighborDecrease | napalm | Trigger which occurs when a device's LLDP neighbors  |
|                           |        | decrease                                        |
| napalm.LLDPNeighborIncrease | napalm | Trigger which occurs when a device's LLDP neighbors  |
|                           |        | increase                                        |
+---------------------------+--------+-------------------------------------------------+
```

Код реализации сенсора LLDP отвечает за определение момента изменения счетчика соседних узлов и активизации соответствующего триггера.

Tlgm: @it_boooks

Рассмотрим, как можно вызвать срабатывание одного из этих триггеров. Регистрируемся на наблюдаемом сетевом устройстве и убеждаемся в том, что существует по меньшей мере один соседний узел LLDP:

```
root@vsrx01> show lldp neighbors
Local Interface    Parent Interface    Chassis Id          Port info          System Name
ge-0/0/1.0         -                   4c:96:14:10:01:00   ge-0/0/2.0         vsrx02
```

Поскольку обнаружен соседний узел на устройстве `ge-0/0/1`, можно отключить этот интерфейс для удаления его из таблицы.

```
root@vsrx01# set interfaces ge-0/0/1 unit 0 disable

[edit]
root@vsrx01# commit
commit complete
```

Этот конкретный сенсор периодически отправляет в журнал некоторые информативные сообщения о счетчике соседних устройств.



Файл журнала для фиксации всех действий сенсоров обычно размещен в каталоге /var/log/st2/st2sensorcontainer.log, но место его расположения можно изменить на любое другое, определив его в файле конфигурации.

```
2017-07-04 23:54:16,134 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 1
2017-07-04 23:54:22,732 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 1
2017-07-04 23:54:28,701 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
went DOWN to 0
2017-07-04 23:54:34,748 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 0
```

Здесь можно видеть, что сенсор точно знает о том, что изначально в счетчике соседних узлов оставалось постоянным значение 1 до тех пор, пока оно не уменьшилось до 0, и это состояние теперь также остается неизменным.

Как уже отмечалось ранее, триггер представляет собой важную часть системы. Можно обратиться к системе StackStorm с запросом списка «экземпляров триггеров», представляющим особые случаи срабатывания триггеров до текущего момента. При каждом уменьшении счетчика соседних узлов для наблюдаемого устройства должен срабатывать триггер `napalm.LLDPNeighborDecrease`. Это можно проверить, чтобы убедиться в правильном функционировании триггера:

```
vagrant@st2vagrant:~$ st2 trigger-instance list --trigger=napalm.LLDPNeighborDecrease
+--------------------------+----------------------------+-----------------------------+
| id                       | trigger                    | occurrence_time  | status   |
+--------------------------+----------------------------+------------------+----------+
| 595c2ab4c4da5f035af38903 | napalm.LLDPNeighborDecrease | < truncated >    | processed |
+--------------------------+----------------------------+-----------------------------+
```

Аналогичным образом извлекаются подробности выполнения операций, для этого можно воспользоваться идентификатором конкретного экземпляра

Tlgm: @it_boooks

триггера, чтобы увидеть информацию о рабочей нагрузке при обработке соответствующего события:

```
vagrant@st2vagrant:~$ st2 trigger-instance get 595c2ab4c4da5f035af38903
+-----------------+-----------------------------------------------+
| Property        | Value                                         |
+-----------------+-----------------------------------------------+
| id              | 595c2ab4c4da5f035af38903                      |
| trigger         | napalm.LLDPNeighborDecrease                   |
| occurrence_time | 2017-07-04T23:54:28.713000Z                   |
| payload         | {                                             |
|                 |     "device": "vsrx01",                       |
|                 |     "timestamp": "2017-07-04 23:54:28.701351",  |
|                 |     "oldpeers": 1,                            |
|                 |     "newpeers": 0                             |
|                 | }                                             |
| status          | processed                                     |
+-----------------+-----------------------------------------------+
```

В следующем разделе рассматриваются правила в системе StackStorm.

Правила

Напомним, что к настоящему моменту были подробно рассмотрены операции, которые позволяют выполнять несложную автоматизацию ежедневных действий, а также сенсоры и триггеры, предназначенные для обработки событий в системе StackStorm. Для реализации автоматизации, управляемой событиями, необходим способ связывания событий, генерируемых триггерами, с операциями или рабочими потоками. Такой способ в системе StackStorm предоставляют правила.

Правила (*rules*) – это определения (как и многие другие объекты в StackStorm) в YAML-файлах. Версия правила на простом английском языке может выглядеть следующим образом: «When X happens, do Y.» («Если произошло событие X, то выполнить Y.»). В нашем случае X – экземпляр триггера (соответствующее событие), а Y – операция или рабочий поток.

Одним из простейших примеров правил является многократное выполнение команды `echo` с интервалом в несколько секунд. В системе StackStorm существует специальный триггер `core.st2.IntervalTimer`, который позволяет интерпретировать завершение заданного интервала времени как событие, требующее ответной реакции:

```
---
name: sample_rule_with_timer
pack: "examples"
description: Sample rule using an Interval Timer.
enabled: true

trigger:
  parameters:
    delta: 5
```

Tlgm: @it_boooks


```
    unit: seconds
  type: core.st2.IntervalTimer

criteria: {}

action:
  parameters:
    cmd: echo "{{trigger.executed_at}}"
  ref: core.local
```

В каждом правиле необходимо выделить три основных элемента:

- `trigger` – в этом разделе определяется имя триггера, который необходимо отслеживать (в нашем примере это триггер `core.st2.IntervalTimer`), а также все параметры, требуемые для этого триггера;
- `criteria` – ограничения, по которым будет определяться соответствие конкретных экземпляров триггера данному правилу. Поскольку здесь не заданы никакие условия, все экземпляры триггера `core.st2.IntervalTimer` подходят для данного правила;
- `action` – здесь определяется операция или рабочий поток, который должен быть выполнен, а также все необходимые параметры. При появлении экземпляра триггера, соответствующего условиям двух первых разделов, будет выполнена указанная здесь операция или рабочий поток.

```
vagrant@st2vagrant:~$ st2 execution list -a id action.ref context.user status
+--------------------------+------------------+-------------+-----------------------+
| id                       | action.ref       | context.user | status                |
+--------------------------+------------------+-------------+-----------------------+
|   595c3093c4da5f035af38bb0 | core.local       | stanley      | succeeded (1s elapsed)  |
|   595c3098c4da5f035af38bb6 | core.local       | stanley      | succeeded (1s elapsed)  |
|   595c309dc4da5f035af38bbc | core.local       | stanley      | succeeded (1s elapsed)  |
+--------------------------+------------------+-------------+-----------------------+
vagrant@st2vagrant:~$ st2 execution get 595c309dc4da5f035af38bbc
id: 595c309dc4da5f035af38bbc
status: succeeded (1s elapsed)
parameters:
  cmd: echo "2017-07-05 00:19:41.779821+00:00"
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: '2017-07-05 00:19:41.779821+00:00'
  succeeded: true
```

Явный признак того, что эти действия были инициированы правилом из приведенного выше примера, – операции выполняются встроенным в систему пользователем `stanley`, а не пользователем, который зарегистрировался в системе (в нашем примере это `st2admin`).

Можно применить правило для оповещения через Slack об уменьшении значения счетчика соседних узлов LLDP:

Tlgm: @it_boooks


```
---
name: "lldp_notify"
pack: "napalm"
enabled: true
description: "Notify of LLDP Neighbor Decrease"

trigger:
  type: "napalm.LLDPNeighborDecrease"
  parameters: {}

criteria: {}

action:
  ref: slack.post_message
  parameters:
    message: "WARNING: {{trigger.device}}'s LLDP Neighbors just went DOWN to
      {{trigger.newpeers}} (was {{ trigger.oldpeers }})""
    channel: '#general'
```

В этом примере отслеживаются все экземпляры триггера `napalm.LLDPNeighborDecrease`. При появлении одного из экземпляров (возникновении события) активизируется операция `slack.post_message` для передачи сообщения в Slack (отметим, что в описании рабочего содержимого используются поля `device`, `newpeers` и `oldpeers` для добавления контекстных данных в сообщение).

Разумеется, можно было бы сделать больше, нежели простое оповещение. Предположим, что количество соседних узлов LLDP уменьшается, потому что кто-то зарегистрировался на отслеживаемом устройстве и случайно отключил один из интерфейсов (подобно тому, как это было сделано в разделе о триггерах). Для того чтобы вернуть этот интерфейс в рабочий режим, вероятнее всего, потребуется принудительная передача на устройство следующей конфигурации:

```
interfaces {
    ge-0/0/1 {
        unit 0 {
            enable;
        }
    }
}
```

Тогда вместо простого оповещения можно определить ответное действие на это событие. Действие состоит в автоматической передаче вышеуказанной конфигурации с помощью операции `loadconfig` из пакета `napalm`:

```
---
name: "lldp_remediate"
pack: "napalm"
enabled: true
description: "Bring interface back up when LLDP neighbor count decreases"

trigger:
  type: "napalm.LLDPNeighborDecrease"
  parameters: {}
```

Tlgm: @it_boooks


```
criteria: {}

action:
  ref: napalm.loadconfig
  parameters:
    hostname: "{{ trigger.device }}"
    config_file: /vagrant/remediation_config.txt
```

Как уже отмечалось выше, активные (выполняющиеся) рабочие потоки по своей функциональности очень похожи на выполняющиеся операции. Можно создать полнофункциональный рабочий поток для устранения неисправностей, автоматического восстановления, передачи оповещений или для объединенной реализации всех этих действий. Правила способны выполнять как простые операции, так и комплексные многофункциональные рабочие потоки.

Краткий итоговый обзор системы StackStorm

О системе StackStorm можно было бы сказать намного больше, но даже рассмотренных выше основных концепций и методик вполне достаточно, чтобы приступить к реализации процессов автоматического восстановления и автоматизации сети, управляемой событиями. Главный принцип системы StackStorm – предоставить пользователю набор инструментов, позволяющих формировать процессы, независимо реагирующие на события в сетевой инфраструктуре, теми же способами, которые ранее выполнялись вручную, но с обеспечением большей надежности.

Более подробную информацию о системе StackStorm можно получить из официальной документации (https://docs.stackstorm.com/), а также из канала сообщества Slack (https://stackstorm.com/?#community), где каждый пользователь может задать любой вопрос и получить квалифицированный ответ.

Резюме

В этой главе рассматривались инструментальные средства автоматизации – Ansible, Salt и StackStorm, а также возможности и методики их практического применения в различных вариантах автоматизации сети. Были приведены многочисленные примеры использования этих инструментов для автоматизации сети, в которых демонстрировались преимущества и недостатки каждой системы.

Tlgm: @it_boooks

Глава 10

Непрерывная интеграция

В этой главе направление нашего учебного курса немного изменится. До настоящего момента в книге излагалась подробная информация о конкретных инструментальных средствах и технологиях, предлагаемых для изучения с целью практического применения их в процессе автоматизации сети. Но было бы ошибкой предполагать, что автоматизация сети – это только новейшие мощные инструменты, – в действительности они представляют лишь фрагмент общей, более сложной картины.

Поэтому основной темой данной главы является оптимизация процессов, связанных с управлением сетью и сетевыми операциями. Имея солидный запас знаний о конкретных инструментальных средствах и технологиях, рассмотренных в предыдущих главах, вы сможете воспользоваться этой главой как руководством к практическому применению инструментов и технологий для решения реальных, действительно сложных задач, с которыми постоянно сталкиваются сетевые операторы и администраторы любых уровней. В этой главе вы найдете ответы на следующие вопросы:

- как можно использовать автоматизацию сети для создания более стабильной и более надежной сетевой среды?
- каким образом можно способствовать увеличению скорости работы сетевой среды в соответствии с требованиями бизнес-деятельности предприятия, но без ущерба для доступности сети?
- какими типами программного обеспечения или инструментальных средств можно воспользоваться, чтобы существенно улучшить процессы, работающие в сетевой среде?

Сетевая отрасль так или иначе связана с любой другой областью информационных технологий (ИТ), поэтому всевозможные простои, изменения стратегии или помехи, влияющие на эффективность процесса, будут оказывать отрицательное воздействие на каждую технологию, связанную с сетевой средой. В наше время такие воздействия проявляются практически в каждой отрасли технологии. В прочих областях ИТ и в бизнес-деятельности сетевая среда в целом воспринимается как некая сущность, которая должна «не путаться под ногами» и «просто работать». Сегодня сеть должна быть доступной всегда, стать более гибкой и более быстрой, чем раньше, а также обеспечивать полную поддержку любого сервиса или приложения, необходимого для бизнеса.

Tlgm: @it_boooks

В действительности универсального средства «на все случаи жизни» не существует, для достижения перечисленных выше целей требуется дисциплина (как в технологическом, так и в общечеловеческом смысле), кроме того, необходимо отказываться от существующих привычных процессов и принципов обмена информацией. Это также связано с большим объемом работы, временем на обучение и освоение новых инструментальных средств. Такой подход к работе на начальном этапе может показаться всего лишь увеличением уровня сложности, но со временем вы поймете, что этот подход позволил увеличить стабильность и скорость сетевых процессов и операций.

Одной из основных идей такой методики является исключение человека из процесса прямого управления сетевой средой. Скептическое отношение к этой идее может быть вполне оправданным, поскольку имеется в виду процесс автоматизации с отстранением человека от работы на длительный интервал времени. Тем не менее исключение человека из процесса прямого управления – это не то же самое, что полное исключение человеческой деятельности. В настоящее время люди осуществляют прямое управление сетями с помощью операций, выполняемых вручную и проходящих по цепочке специалистов, для внесения изменений в сетевую среду, как показано на рис. 10.1.

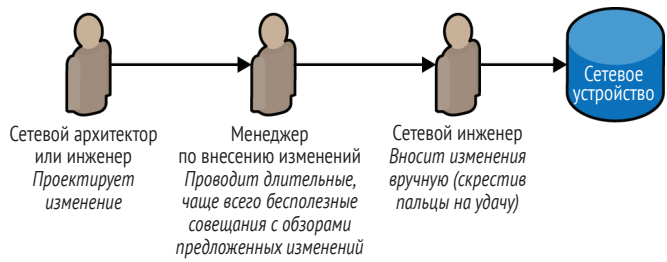

Сетевое устройство
Сетевой архитектор или инженер
Проектирует изменение
Менеджер по внесению изменений
Проводит длительные, чаще всего бесполезные совещания с обзорами предложенных изменений
Сетевой инженер
Вносит изменения вручную (скрестив пальцы на удачу)

Рис. 10.1 ❖ Люди в цепочке прямого управления сетью

Эта методика уже зарекомендовала себя как медленная и чрезвычайно трудозатратная, к тому же она практически не обеспечивает надежность при внесении изменений в сетевую среду. Такой подход чаще всего только мешает работе, хотя и создает видимость безопасной процедуры внесения изменений.

При обсуждении исключения людей из процесса прямого управления речь идет о непрерывной интеграции (Continuous Integration), то есть об автоматизации отдельных задач, которые должны выполняться в ходе управления изменениями инфраструктуры, и освобождении технических ресурсов, связанных с этим конвейерным процессом, тем самым усовершенствуя его и повышая эффективность (см. рис. 10.2).

В результате такого крупномасштабного перехода к непрерывной интеграции можно действительно обеспечить защиту от человеческих ошибок при выполнении сетевых операций, заменив исторически сформировавшийся механизм внесения изменений, управляемый людьми.

Tlgm: @it_boooks

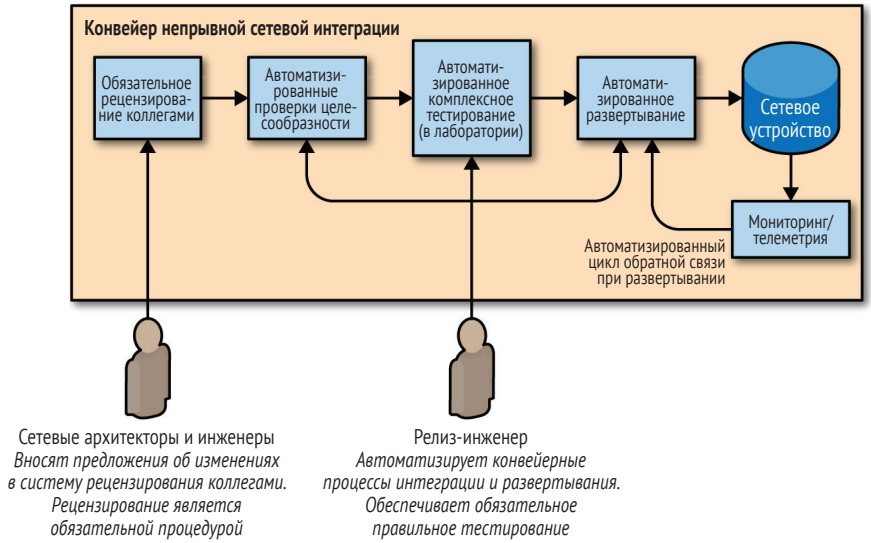

Конвейер непрывной сетевой интеграции
Обязательное рецензирование коллегами
Автоматизированные проверки целесообразности
Автоматизированное комплексное тестирование (в лаборатории)
Автоматизированное развертывание
Сетевое устройство
Мониторинг/ телеметрия
Автоматизированный цикл обратной связи при развертывании
Сетевые архитекторы и инженеры
Вносят предложения об изменениях в систему рецензирования коллегами. Рецензирование является обязательной процедурой
Релиз-инженер
Автоматизирует конвейерные процессы интеграции и развертывания. Обеспечивает обязательное правильное тестирование

Рис. 10.2 ❖ Автоматизированное внесение изменений с применением непрерывной интеграции

Важные предпосылки

Для наиболее успешного применения концепций, рассматриваемых в этой главе, необходимо помнить о нескольких основных принципах, описанных в следующих подразделах.

Чем проще, тем лучше

Самое лучшее, что можно сделать для подготовки конкретной сетевой среды к автоматизации, – не увлекаться изучением программного кода и применением новейших разрекламированных инструментальных средств автоматизации, поскольку все это относится к разработке проектного решения сети. Избегайте излишне усложненных решений и структур и всегда старайтесь развертывать сетевые сервисы по простым стандартным методикам.

Другими словами, можно принять решение о развертывании сетевых конфигураций, управляемом шаблонами. Если для каждого сетевого устройства предназначена собственная специализированная конфигурация с разнообразными функциональными возможностями, то будет чрезвычайно трудно создать шаблоны для большой группы таких устройств.



Шаблоны сетевой конфигурации подробно рассматривались в главе 6.

Чем более продуманной будет логическая согласованность и целостность сетевого проектного решения, тем меньшими будут трудозатраты, когда при-

Tlgm: @it_boooks

дет время автоматизации этой сети. Чаще всего это означает независимость от функциональных особенностей устройств от различных производителей или полный отказ от специфических встроенных функциональных возможностей и реализацию сетевых сервисов исключительно на программном уровне.

Люди, процесс и технология

В предыдущих главах мы рассматривали множество замечательных технологий и инструментальных средств, но в наше время в сетевой отрасли существуют гораздо более серьезные и сложные задачи – проблемы организации рабочего процесса и совместной работы с другими ИТ-группами, которые, возможно, не обладают знаниями и умениями, соответствующими вашему уровню.

Многие главы этой книги посвящены специальным технологиям и инструментальным средствам, которые можно использовать для создания эффективных систем автоматизации сети. В настоящее время предлагается большое количество методик, применяемых для решения задач автоматизации, многие из них, вероятно, являются новыми для большинства сетевых инженеров, поэтому важно знать об их существовании. Кроме того, важно усовершенствовать и изменять способы обмена информацией с другими областями информационных технологий и сферой бизнеса в целом – об этом пойдет речь в главе 11.

Но в текущей главе мы будем рассматривать некоторые усовершенствования рабочего процесса, которые разработчики программного обеспечения уже используют в течение некоторого времени для улучшения методики внесения изменений в приложения. Конечная цель такой методики – быстрое внесение изменений и ввод их в эксплуатацию с минимизацией риска отрицательного воздействия. Сообщество сетевых инженеров при изучении этого процесса может извлечь немало весьма важных уроков, особенно в плане решения задачи автоматизации сети.

Изучение программного кода

Сразу же следует отметить, что совсем не обязательно становиться разработчиком программного обеспечения, чтобы применять концепции, изложенные в этой главе. В действительности в текущей главе поставлена задача донесения основной идеи до читателя. Тем не менее, вероятнее всего, вы вскоре обнаружите, что ни одно инструментальное средство (и даже набор инструментальных средств) не способно решить все ваши проблемы.

Возможно, вам придется ликвидировать некоторые пробелы в знаниях о непрерывной интеграции и приобретать практические навыки посредством написания некоторых вариантов специализированных решений, таких как скрипты. Используйте это как возможность расширения области ваших знаний и умений. Рассматриваемый в главе 4 Python является вполне подходящим языком для начинающих, он достаточно прост для быстрого изучения и достаточно надежен для решения разнообразных сложных задач.

Tlgm: @it_boooks

Введение в непрерывную интеграцию

Прежде чем начать подробное изучение методики непрерывной интеграции и ее применимости к процессу автоматизации сети, следует сначала обратиться к ее истокам, чтобы понять, какую пользу непрерывная интеграция приносит группам разработки программного обеспечения.

В первую очередь при обсуждении реализации непрерывной интеграции следует обратить особое внимание на достижение двух главных целей:

- *ускоренная реакция* – способность более быстро реагировать на необходимость изменений, требуемых по условиям бизнес-деятельности;
- *улучшение надежности* – использовать предыдущий практический опыт и улучшать качество и стабильность всей системы в целом.

До внедрения методики непрерывной интеграции изменения в программное обеспечение чаще всего вносились в виде крупных пакетов «заплаток» и дополнений. При этом иногда разработчикам требовалось несколько месяцев, чтобы ввести новые и исправленные функциональные возможности в реальную эксплуатацию. Такой подход формировал чрезвычайно длительные циклы обратной связи, и если возникали какие-либо серьезные проблемы или вводились новые требования/функциональные возможности, то для решения таких проблем тратилось очень много времени. Подобная неэффективность не только существенно задерживает разработку и внедрение новых функциональных возможностей, но также ухудшает качество программного обеспечения.

Кажется вполне очевидным и естественным, что было бы неплохо, если бы разработчики могли просто продвигать вносимые изменения в ПО непосредственно в процессе эксплуатации. Это решило бы проблему скорости – разработчики смогли бы сразу увидеть результаты внесения изменений. Но такая методика связана с чрезвычайно высокой степенью риска. В предлагаемой модели высока вероятность внесения ошибок в реально эксплуатируемое ПО, а в итоге это может привести к серьезным негативным последствиям для любой бизнес-деятельности.

Непрерывная интеграция (в совокупности с непрерывным развертыванием (Continuous Deployment), которое мы рассмотрим ниже в текущей главе) представляет оптимальное решение для обеих сфер деятельности. В этой модели изменения продвигаются в эксплуатацию очень быстро, но эта процедура выполняется в комплексе с обязательным тестированием и тщательной проверкой вносимых изменений для большей уверенности в том, что внедрение изменений не вызовет проблем в режиме реальной эксплуатации.

В следующем разделе рассматриваются некоторые основные концепции и компоненты непрерывной интеграции, затем объясняется, как можно применить эти концепции к процессу автоматизации сети.

Основы непрерывной интеграции

Если говорить кратко, то непрерывная интеграция – это комплекс возможностей внесения изменений в репозиторий исходного кода в любое время.

Tlgm: @it_boooks

Группа разработчиков вне зависимости от того, где и когда работают ее участники, может «интегрировать» изменения в некоторый совместно используемый репозиторий в любое время, поскольку располагает инструментальными средствами, позволяющими всей группе точно знать (все инструменты автоматизированы), что вносимые изменения не приведут к нарушению функциональности всей системы в целом.

Возможно, вам знаком термин *«конвейер»* (*pipeline*), который используется при обсуждении методики непрерывной интеграции. Причина возникновения этого термина заключается в том, что непрерывная интеграция представляет собой не одну особенную технологию, а обычно является комплексом различных инструментальных средств и технологий, совместно используемых для достижения основной цели. Изменения в кодовой базе продвигаются через эти инструментальные средства с предварительно определенной последовательностью операций, и весь комплекс в итоге формирует *конвейер непрерывной интеграции* (*CI pipeline*). Все изменения обязательно должны пройти через конвейер непрерывной интеграции, прежде чем будут переданы для развертывания в эксплуатационной среде (рис. 10.3).

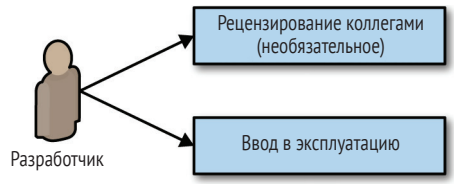


Рис. 10.3 ❖ Развертывание программного обеспечения непосредственно в эксплуатационной среде

Схема на рис. 10.3 должна быть вам хорошо знакома. Практически все сетевые инженеры действуют именно так почти всегда. Возможно, сетевые инженеры не занимаются развертыванием программного обеспечения, но они, несомненно, выполняют развертывание критических изменений в инфраструктуре, которые обычно имеют более обширную «область поражения», чем большинство прочих изменений инфраструктуры. Регистрация в сеансе через SSH-соединение на маршрутизаторе для внесения каких-либо изменений в конфигурацию не менее рискованна, чем редактирование исходного кода приложения, находящегося в режиме эксплуатации.

В противоположность такому подходу методика непрерывной интеграции предлагает только один пункт, в котором человек может вносить изменения, – это конвейер непрерывной интеграции, точнее, самая первая часть этого конвейера, которая называется *рецензирование коллегами* (*peer review*). Это первый шаг в длинной последовательности автоматизированных этапов, таких как автоматизированное тестирование и проверка целесообразности и обоснован-

Tlgm: @it_boooks

ности вносимых изменений. Для того чтобы изменения действительно были введены в эксплуатацию, они обязательно должны пройти весь конвейер полностью без каких-либо исключений (см. рис. 10.4).

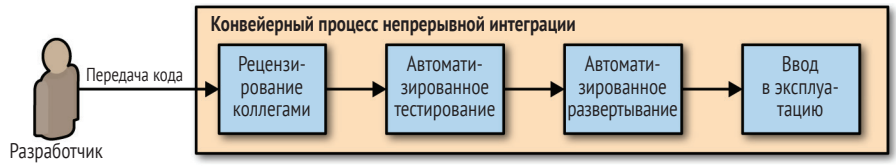


Рис. 10.4 ❖ Развертывание программного обеспечения в конвейере непрерывной интеграции

И с этой точки зрения методика непрерывной интеграции увеличивает скорость и в то же время позволяет сохранить (или даже улучшить) стабильность. Такой подход возможен благодаря созданию единственной точки входа для внесения любых изменений. Становится неоптимальным изменение инфраструктуры, которое не прошло весь набор тестов, гарантирующих прохождение технического рецензирования коллегами, а также позволяющих избежать повторения старых ошибок.



Некоторые организации нанимают специалистов, которых называют релиз-инженерами (release engineers), для управления конвейером непрерывной интеграции. Релиз-инженеры в совершенстве владеют такими инструментальными средствами, как Git, инструменты тестирования, серверы сборки ПО и системы коллегиального рецензирования. Они обеспечивают целостность и согласованность всего конвейера, поэтому с разработчиков снимается эта обязанность. Конечная цель релиз-инженеров – полная автоматизация процесса от рабочего компьютера разработчика до внедрения в эксплуатацию (именно поэтому они называются «релиз»-инженерами).

Не так-то просто определить точное количество требуемых релиз-инженеров, но при любой возможности, особенно если ваша группа достаточно велика для обоснования такой необходимости, рекомендуется воспользоваться услугами релиз-инженеров.

Мы рассмотрели основы непрерывной интеграции, после чего можно перейти к более подробному изучению некоторых ее компонентов и связанных с ними концепций и технологий, которые необходимы для дальнейшего освоения этой методики.

Непрерывная доставка

Непрерывная доставка (Continuous Delivery – CD) – еще один термин, с которым вы, вероятнее всего, уже встречались и который тесно связан с методикой непрерывной интеграции. Непрерывная доставка – это концепция, в соответствии с которой группа разработки ПО непрерывно предоставляет программное обеспечение с возможностью его непосредственного развертывания в эксплуатацию. Таким образом, выполняется *доставка* (*delivering*) гото-

Tlgm: @it_boooks

вого к работе программного обеспечения в форме полностью подготовленной к развертыванию кодовой базы.



Непрерывное развертывание (Continuous Deployment) обычно предполагает, что разработчики всегда передают новый код в эксплуатацию немедленно, без каких-либо задержек. В ИТ-отрасли лишь недавно вместо этого термина стал использоваться термин «непрерывная доставка» (Continuous Delivery), который означает, что ваш код всегда находится в состоянии, позволяющем развертывать его в любой момент, но делать это не обязательно. Организация может принять план и расписание процедур развертывания, например в ночное время или в определенный день недели.

Методику непрерывной интеграции можно относительно просто применить для автоматизации сети (как вы сами убедитесь в следующих разделах), но непрерывная доставка требует несколько более тщательного планирования. В оставшейся части этой главы граница между непрерывной интеграцией и непрерывной доставкой может показаться слишком неопределенной, расплывчатой с точки зрения автоматизации сети, поэтому всегда необходимо помнить о следующем:

- что именно развертывается;
- для чего/кого выполняется развертывание.

Это очень важные вопросы, поскольку ответы на них определяют вашу модель развертывания. Например, некоторые группы обслуживания сети могут выполнять всю работу по автоматизации с помощью собственных приложений на языке Python. Это относительно простое решение, так как в составе группы сопровождения сетевой инфраструктуры фактически существует группа разработки ПО.

С другой стороны, существует классический пример автоматизации сети: передача некоторого типа объекта конфигурации (например, некоторого YAML-файла) в репозиторий Git и организация конвейера непрерывной интеграции/непрерывной доставки, в котором выполняется ряд основных проверок целесообразности и обоснованности переданного объекта до конечного этапа его передачи в инструментальное средство, такое как Ansible. В результате изменения своевременно, без лишних задержек вводятся в эксплуатацию на сетевых устройствах. Это может успешно работать в некоторых организациях, но такой подход аналогичен быстрому развертыванию группой разработки ПО каждого исправления («заплатки») в программе и немедленному вводу его в эксплуатацию, это требуется далеко не всегда.

Возможно, следует рассмотреть вариант среды с «поэтапной» организацией, в которую такие изменения могут доставляться непрерывно, и когда в бизнес-деятельности требуется окончательное внедрение этих изменений в эксплуатацию, их можно переместить с предварительного подготовительного этапа, на котором они (предположительно) были тщательно протестированы. Во время написания этой главы многие производители сетевого оборудования уже знали о наших запросах о предоставлении виртуальных образов их платформ для существенного упрощения реализации вышеописанной методики.

Tlgm: @it_boooks



Все упомянутые выше виртуальные устройства вполне пригодны для тестирования процесса автоматизации, но не все в действительности способны воспроизводить реальный эксплуатационный сетевой трафик. Для получения более подробной информации об этой проблеме обращайтесь к документации конкретного производителя оборудования.

Также необходимо хорошо продумать процедуры отката (rollback), то есть возврата к предыдущему корректному состоянию. Как часто вы берете конфигурации из «эксплуатационного» репозитория Git и используете их для перезаписи текущих «рабочих» конфигураций или, по крайней мере, сравниваете эти две версии? Если такая процедура выполняется крайне редко или вообще игнорируется, то даже если производится откат репозитория, то для версий этих конфигураций, находящихся в эксплуатации, вероятнее всего, откат не будет выполнен. Каким будет воздействие отката репозитория Git, если используются Ansible, или Puppet, или какие-то специализированные Python-программы? Необходимо тщательно исследовать и взять под контроль этот уровень программного стека, а также хорошо понимать, как применяемые инструментальные средства и ПО будет реагировать (или не реагировать) на процедуры отката текущих эксплуатационных конфигураций.

В действительности вы сами должны ответить на вопрос о необходимости применения методики непрерывной доставки. То, что работает в одной организации, может оказаться непригодным в другой из-за большого разнообразия инструментальных средств и языков, доступных для решения задач автоматизации сети. Тем не менее эта глава, по крайней мере, должна определить некоторые отправные пункты и идеи относительно того, как правильно организовать доставку и внедрение изменений в сети средствами автоматизации.

Разработка через тестирование

Не менее важной является еще одна парадигма разработки ПО, которая становится все более распространенной, – разработка через тестирование (Test-Driven Development – TDD).

Предположим, что вы являетесь разработчиком ПО и перед вами поставлена задача создания некоторой новой функциональной возможности в текущем проекте. Разумеется, сначала вы занимаетесь сбором основных требований, на их основе формируете предварительное проектное решение, затем приступаете к созданию требуемой функциональности (см. рис. 10.5). Можно даже говорить о том, что вы применяете методику непрерывной интеграции, поскольку в дальнейшем создаются некоторые модульные тесты, проверяющие создаваемую функциональность. К сожалению, работа не всегда выполняется по такому плану. В действительности при создании тестов после полной реализации функциональной возможности часто очень трудно подобрать необходимый набор, а в самых крайних случаях тесты считают менее важными, чем сама реализация функциональности.

Tlgm: @it_boooks

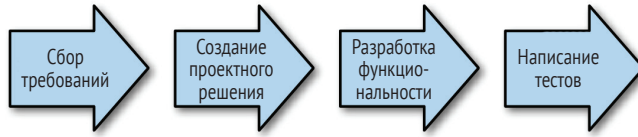


Рис. 10.5 ❖ Жизненный цикл разработки ПО до применения методики разработки через тестирование

На практике такой подход быстро приводит к накоплению «технических задолженностей» (technical debt). Другими словами, если не создавать тесты в первую очередь, то всегда возникает искушение отказаться от них сразу после завершения разработки требуемой функциональной возможности или вообще забыть о тестировании. Это неизбежно ведет к разрыву тестового покрытия, а в крупных проектах этот разрыв нарастает со временем.

Разработка через тестирование возникла как средство решения этой проблемы. При использовании методики разработки через тестирование после этапа сбора требований и формирования на их основе предварительного проектного решения в первую очередь обязательно пишется тест для проектируемой функциональной возможности до начала ее реализации (см. рис. 10.6). Разумеется, тест должен провалиться, так как отсутствует тестируемый программный код. Таким образом, итоговая проверка требуемой функциональной возможности заключается в написании такого программного кода, который успешно пройдет предварительно созданный тест (или набор тестов).

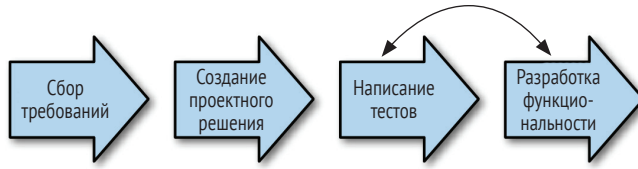


Рис. 10.6 ❖ Жизненный цикл разработки ПО после внедрения методики разработки через тестирование

Почему необходимо использовать методику разработки через тестирование? Вы сразу же получаете основное преимущество – сокращение «технической задолженности». Если тесты создаются до начала реализации функциональности, то не возникает соблазн оставить тестовое покрытие без внимания, отдавая безусловный приоритет замечательным новым функциям. Но здесь также возникает небольшое концептуальное различие. При написании тестов на предварительном этапе разработчик обязательно должен полностью понимать, как будет использоваться создаваемое ПО, так как тесты пишутся для обеспечения правильной работы программ. Нет никаких сомнений, что это оказывает положительное влияние на качество ПО.

Tlgm: @it_boooks

Когда описанные выше концепции применяются к процессу автоматизации сети, сначала следует провести несколько параллелей или аналогий. Сетевая среда является бизнес-ресурсом в той же степени, что и приложения, работающие в сетевой среде. Следовательно, весьма важно обеспечить надлежащее тестирование, которое не только проверяет корректность любых изменений, вносимых в сетевую среду, но также помогает заблаговременно узнать о любых проблемах (планирование пропускной способности). Вы собираете подробную статистику о приложениях, работающих в сети? Вероятно, не только с помощью сервиса SNMP на сетевых устройствах – необходимы данные от самих программных приложений. Сетевой инженер обязательно должен перенять полезный опыт разработчиков ПО и оценить важность практических вариантов использования, а также практический опыт пользователей.

В этой главе методика разработки через тестирование рассматривается с двух точек зрения. Необходимо напомнить о двух фактах, которые важны как при использовании существующих инструментальных средств автоматизации сети, так и при написании собственного программного обеспечения:

- мы уделяем достаточно внимания качеству системы автоматизации сети, обеспечению длительного времени непрерывной работы и положительному практическому опыту наших пользователей и приложений, чтобы гарантировать, что система правильно тестируется (при этом – далеко не в минимальном объеме);
- тестирование настолько важно, что должно выполняться до начала автоматизации какого-либо отдельного объекта в сетевой среде. С доступным на текущий момент программным обеспечением и при постоянном снижении уровня сложности освоения этих инструментальных средств для тех, кто не занимается разработкой профессионально, нет никаких оправданий отказа от тестирования.



В следующих разделах вы получите более подробную информацию о том, как организовать тестирование, при этом приведенные выше два факта могут показаться слегка надуманными. Поэтому сейчас просто запомните эти два факта – они чрезвычайно важны для достижения успеха в любом процессе автоматизации сети.

Применяя методику, подобную разработке через тестирование, мы не только способствуем быстрому внедрению программных приложений, но также формируем воспроизводимый процесс, в ходе которого обеспечивается постоянная уверенность в том, что сетевая среда полностью удовлетворяет потребности приложения, вне зависимости от изменений, вносимых в конфигурацию или в саму сетевую среду. Далее в текущей главе будут рассматриваться некоторые специализированные инструментальные средства и технологии, которые можно использовать для достижения этих целей.

Tlgm: @it_boooks

Применимость методики непрерывной интеграции к сетевой среде

До сих пор рассматривались такие концепции, как непрерывная интеграция и разработка через тестирование, оценивались важность и полезность этих концепций для групп разработки ПО. Но теперь мы будем применять эти и некоторые другие концепции исключительно к процессу автоматизации сети, поскольку это основная тема нашей книги.

Зачем это нужно? Какую пользу непрерывная интеграция и разработка через тестирование могут принести сетевым инженерам? Напомним главные цели методики непрерывной интеграции:

- *ускорение процесса* – способность обеспечить более быструю ответную реакцию на необходимость изменений, диктуемых бизнес-деятельностью;
- *повышение надежности* – извлечение полезных уроков из ранее проделанной работы, улучшение качества и стабильности всей системы в целом.

Эти цели, достижение которых дает весомые результаты для написания более стабильных программ и формирования более производительных групп разработки ПО, также может не менее эффективно помочь в создании более надежной сетевой среды. Автоматизация, которая пренебрегает любой из двух указанных целей, фактически является бесполезной.

«Непрерывная интеграция для сетевой среды» в основном имеет тот же смысл, что и в классическом примере разработки ПО, – создание единственного пункта, в котором разрешено внесение каких-либо изменений в сетевую инфраструктуру. При этом тестирование и рецензирование вносимых изменений автоматизировано и обязательно.

В течение долгого времени мы воспринимали и администрировали свои сети как черные ящики, некоторым образом соединенные друг с другом. Такой образ мыслей не подходит для применения практических методик и реализации концепций непрерывной интеграции. Поэтому в первую очередь необходимо изменить образ мышления – начать воспринимать сетевую среду как объединенный пул ресурсов и динамически изменяемых конфигураций, то есть как систему с постоянно изменяющимися рабочими средами и требованиями.

Это основная идея процесса перехода к концепции «инфраструктура как код» – сопровождение состояния и конфигурации сетевой инфраструктуры с использованием тех же процессов, которые разработчики ПО применяют для управления исходным кодом.

Tlgm: @it_boooks

Конвейер непрерывной интеграции для сетевой среды

До сих пор мы рассматривали ряд концепций и теоретических положений высокого уровня, на которых основана методика непрерывной интеграции. Теперь необходимо применить все эти концепции на практике. В этом разделе приводятся практические примеры и описываются инструментальные средства, помогающие достичь целей, поставленных в контексте автоматизации сети.

При изучении практических примеров необходимо помнить о следующих фактах:

- инструментальные средства, используемые в текущем разделе, – это всего лишь примеры. В каждой категории инструментов предлагается более богатый выбор, по сравнению с представленным здесь. Рекомендуется самостоятельно опробовать инструментальные средства, доступные в каждой категории, и определить наиболее подходящие для вашей работы;
- этот раздел не случайно размещен после теоретического раздела. Применение предлагаемых инструментальных средств без корректировки неэффективных (зачастую неприемлемых) процессов, которые существуют во многих организациях в течение многих лет, не даст никакого полезного результата.



Следует отметить, что рассматриваемые здесь инструментальные средства могут конфигурироваться разнообразными способами. В текущем разделе представлен только один способ, поэтому необходимо постоянно помнить об основных концепциях и применять правильную конфигурацию, чтобы в полной мере воспользоваться преимуществами инструмента для своей организации.

Ниже перечислены пять основных компонентов конвейера непрерывной интеграции для сетевой среды:

- рецензирование коллегами;
- автоматизация сборки;
- средства развертывания;
- среда тестирования/разработки/перемещения данных;
- инструментальные средства тестирования и автоматизация сети по методике разработки через тестирование.

Для наглядной демонстрации перечисленных концепций в текущей главе используется проект Templatizer, позволяющий обрабатывать шаблоны Jinja, чтобы в результате получить конфигурации сетевых устройств на основе данных из файлов данных, заисанных на языке YAML. Многие примеры активно используют Git-репозиторий Templatizer, размещенный на нашем частном Git-сервере.

На рис. 10.7 приведен полезный наглядный пример для рассматриваемого здесь процесса непрерывной интеграции.

Tlgm: @it_boooks

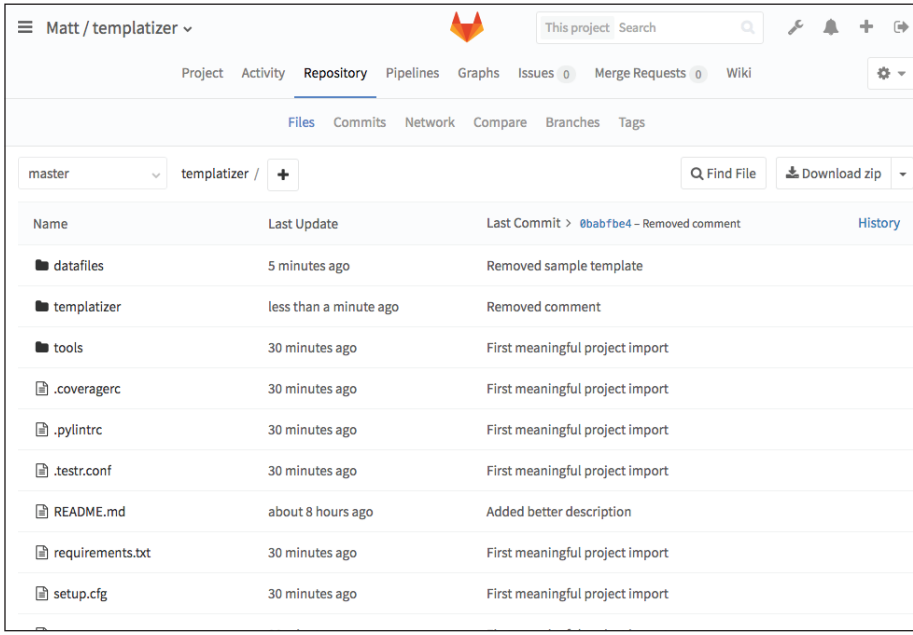


Рис. 10.7 ❖ Проект Templatizer

Рецензирование коллегами

При обсуждении процедуры рецензирования коллегами в обычном смысле применительно к разработке ПО, как правило, речь идет об исходном коде для какого-либо приложения. Разработчик передает фрагмент кода с исправлением («заплатку» – patch), содержащий четко определенные различия в некоторых файлах исходного кода. Затем этот фрагмент-заплатка передается каким-либо способом в систему рецензирования кода, и рецензент (или несколько рецензентов) внимательно изучает предложенный код, дает комментарии по нему или выносит положительный вердикт.

Применение этой части конвейера к процессу автоматизации сети во многом напоминает приведенный выше пример с изменением исходного кода. В некоторых главах книги, например в 5 и 6, рекомендуется использование методики «инфраструктура как код» для автоматизации сети, когда вся информация, относящаяся к конфигурации, интерпретируется так же, как разработчик поступает с исходным кодом. В нашем случае вместо кода на языке Java могут существовать файлы на языке YAML или шаблоны Jinja. Это обычные текстовые файлы, поэтому для них можно организовать автоматическое тестирование по аналогии с тестированием исходного кода.

Знания, полученные о системах управления версиями в главе 8, где описывалась система Git, мы будем использовать не только как основу процесса управления версиями объектов различных конфигураций, такими как YAML-

Tlgm: @it_boooks

файлы, но также как основу реализации первого этапа конвейера непрерывной интеграции – рецензирование коллегами, – чтобы привлечь к процессу дополнительных специалистов, наблюдающих за вносимыми изменениями. Это привносит бóльшую уверенность в том, что все делается правильно.

Если вы занимаетесь сопровождением реально эксплуатируемой ИТ-инфраструктуры любого типа, то, вероятнее всего, регулярно принимаете участие в совещаниях комиссии по утверждению изменений (Change Approval Board – CAB). Возможно, вы являетесь ответственным за заполнение формы, описывающей предлагаемые изменения в конфигурации, затем присутствуете на длительном совещании, где вам предоставлено право сказать несколько слов о корректности и безопасности предложенных изменений, чтобы убедить комиссию принять эти изменения. Этот процесс имеет глубокие корни в современной ИТ-индустрии, но не слишком способствует минимизации рисков или обеспечению прозрачности во взаимодействии взаимосвязанных технических групп. Это устаревший способ работы.

При обсуждении использования методики непрерывной интеграции в сетевой среде мы начинаем с реализации идеи рецензирования коллегами, которая может показаться очень похожей на подход, описанный в предыдущем абзаце, тем не менее между этими методиками имеются существенные различия. При внесении изменений по методике непрерывной интеграции просто создается новая ветвь в репозитории Git и вносится предлагаемое изменение. Поскольку все конфигурации содержатся в репозитории Git, который является частью конвейера непрерывной интеграции, нет необходимости запрашивать специальное разрешение перед выполнением какой-либо работы в отдельной ветви, так как результат этой работы в действительности не будет внедрен в эксплуатацию до тех пор, пока не пройдет рецензирование и не будет включен в основную ветвь репозитория (master).

У этой новой модели есть весьма существенные преимущества. При обязательном рецензировании коллегами нет необходимости «подробно описывать» каждое предлагаемое изменение и надеяться, что все было сделано правильно, когда придет время внедрения этого изменения. Теперь само изменение представляет собой самодостаточное описание. Нет никакой неоднозначности в понимании того, что вы собираетесь делать, потому что это выясняется с помощью системы рецензирования коллегами. Для действительного внедрения предложенного изменения в эксплуатацию рецензент (или рецензенты) просто включает(ют)/объединяет(ют) вашу рабочую ветвь с основной ветвью (master) репозитория.

Для платформ рецензирования кода существует несколько вариантов. Ниже приведен неполный список таких платформ:

- GitHub – широко известный сервис типа SaaS (Software as a Service – программное обеспечение как услуга), предлагающий возможности рецензирования и просмотра исходного кода (также доступна корпоративная версия за отдельную плату);

Tlgm: @it_boooks

- GitLab – вариант, предлагаемый сообществом разработчиков ПО с открытым исходным кодом. Бесплатная загрузка и запуск под защитой локального сетевого экрана (firewall). Также предлагается многоуровневый сервис SaaS и корпоративная версия с возможностью защиты исходного кода;
- Gerrit – сопровождение открытого исходного кода, сложная структура, но с многочисленными возможностями интеграции. Этот вариант выбрали многие проекты с открытым исходным кодом.

Во всех трех вариантах используется Git для реализации компонента управления версиями (таким образом, Git фактически является интерфейсом для передачи исходного кода в репозиторий сервиса). Но поверх Git на этих платформах реализованы собственные специализированные рабочие потоки (процессы) рецензирования исходного кода. Например, на GitHub можно вносить дополнительные изменения простым вводом очередных коммитов в одну и ту же ветвь, но в Gerrit автор изменений всегда должен работать с одним и тем же коммитом (таким образом, для внесения дополнительных изменений требуется флаг `--amend`).

В этой главе будет использоваться сервис GitLab, главным образом потому, что он предлагает большой объем бесплатных функциональных возможностей. Кроме того, для этого сервиса не требуется слишком большого объема работы по настройке. Но следует помнить о том, что другие системы, возможно, окажутся более подходящими для вашей работы.



К настоящему моменту вы должны хорошо знать не только шаблоны Jinja и файлы YAML, но также принципы и приемы практической работы с репозиторием Git. При рассмотрении следующих примеров предполагается, что проект Templatizer уже был клонирован в локальную файловую систему и мы полностью готовы к работе с ним.

В качестве наглядного примера добавим несколько шаблонов Jinja и файлов YAML, чтобы в проекте Templatizer появилась возможность создания конфигураций для интерфейсов сетевых устройств. Начнем с создания новой ветви репозитория Git для выполнения коммитов, соответствующих заявленным изменениям:

```
~$ git checkout -b "add-interface-template"

Switched to a new branch 'add-interface-template'
```

Сейчас мы находимся в новой ветви, существующей только на нашем компьютере (пока еще не выполнена команда git push), которая не является основной (master) ветвью. Поэтому мы просто вносим изменения. Мы не ожидаем какого-либо утверждения (одобрения) перед началом – в первую очередь выполняется работа, которая говорит сама за себя, когда приходит время ее оценки и подтверждения (или отказа).

После добавления шаблона и файла YAML система Git должна выдать оповещение о наличии двух новых файлов, которые пока не отслеживаются:

Tlgm: @it_boooks


```
~$ git status

On branch add-interface-template
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    datafiles/interfaces.yml
    templatizer/templates/interfaces.j2

nothing added to commit but untracked files present (use "git add" to track)
```

Теперь необходимо выполнить коммит и передать его в исходный удаленный репозиторий. В рассматриваемом примере это репозиторий Gitlab, упоминаемый ранее:

```
~$ git add datafiles/ templatizer/

~$ git commit -s -m "Added template and datafile for device interfaces"
 [add-interface-template 4121bfa] Added template and datafile for device interfaces
  2 files changed, 10 insertions(+)
  create mode 100644 datafiles/interfaces.yml
  create mode 100644 templatizer/templates/interfaces.j2

~$ git push origin add-interface-template
 Counting objects: 7, done.
 Delta compression using up to 8 threads.
 Compressing objects: 100% (7/7), done.
 Writing objects: 100% (7/7), 718 bytes | 0 bytes/s, done.
 Total 7 (delta 2), reused 0 (delta 0)
 To http://gitlab/Matt/templatizer.git
  * [new branch]      add-interface-template -> add-interface-template
```

Следующий шаг – регистрация в системе рецензирования кода (GitLab) и инициализация этапа, на котором начинается рецензирование кода коллегами. В каждой системе рецензирования кода существует собственный рабочий поток, но в любом случае все эти потоки выполняют одинаковую работу. Например, в Gerrit используются термины change (изменение) и patchset (набор патчей), а в GitHub соответствующие операции называют pull request (запрос на принятие изменений). В общем, все эти средства предоставляют возможность заявить: «Я предлагаю изменение и хотел бы, чтобы оно было включено в основную ветвь» (как правило, в ветвь master).

В GitLab используется концепция, очень похожая на запрос на принятие изменений в GitHub, – merge request (запрос на объединение ветвей). После того как предлагаемые изменения внесены в отдельную ветвь, в мастере создания запросов на объединение ветвей можно определить предложение о слиянии коммита в ветви add-interface-template с основной ветвью, которая считается стабильной (stable) в данном проекте (см. рис. 10.8).

Tlgm: @it_boooks

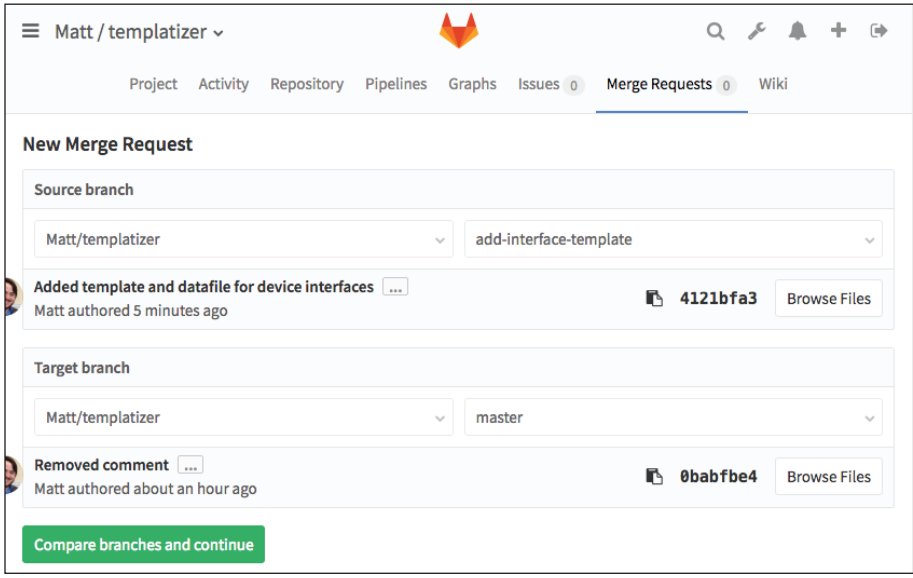


Рис. 10.8 ❖ Создание запроса на объединение ветвей

Далее следует экран, где необходимо подтвердить свои намерения, после чего создается запрос на объединение ветвей. Помните о том, что ваше предложение пока остается лишь запросом. Он все еще не оказывает никакого воздействия на основную ветвь (master) и, соответственно, на текущую стабильную версию проекта Templatizer. Пока это только внесенное нами предложение, которое послужит отправным пунктом для будущего рецензирования коллегами.

Следующий шаг – передача созданного запроса на объединение ветвей на рецензирование авторитетным лицом (или лицами). Эта часть рабочего потока может быть различной для каждой организации, поскольку основана на технической культуре группы. Кроме того, это зависит еще и от платформы рецензирования. Некоторые группы ограничивают доступ к основной ветви, поэтому только определенные представители руководства могут принимать запросы на объединение ветвей. Другие группы используют системы, основанные на профессиональной этике и честном отношении к делу. В этом случае требуется, чтобы каждый запрос на объединение ветвей рецензировался как минимум одним из коллег. Общепринятое соглашение заключается в том, что процедура включения любых изменений в основную ветвь задерживается до того момента, когда один из коллег дает оценку «+1», что фактически означает: «С моей точки зрения это изменение готово к включению в главную ветвь». Это может произойти немедленно, или рецензент может сначала давать какие-либо комментарии или указания, прежде чем принять решение об оценке +1.

Tlgm: @it_boooks

Наш воображаемый коллега Фред (Fred) готов к рецензированию предложенного изменения в проекте Templatizer, и мы можем включить его в этот процесс несколькими способами. Большинство инструментальных средств рецензирования кода предоставляют функцию «добавления» рецензента, который оповещается об этом по электронной почте или можно сообщить ему об этом напрямую. В любом случае, на рис. 10.9 показано, что Фред вовлечен в процесс рецензирования предложенного изменения на GitLab.

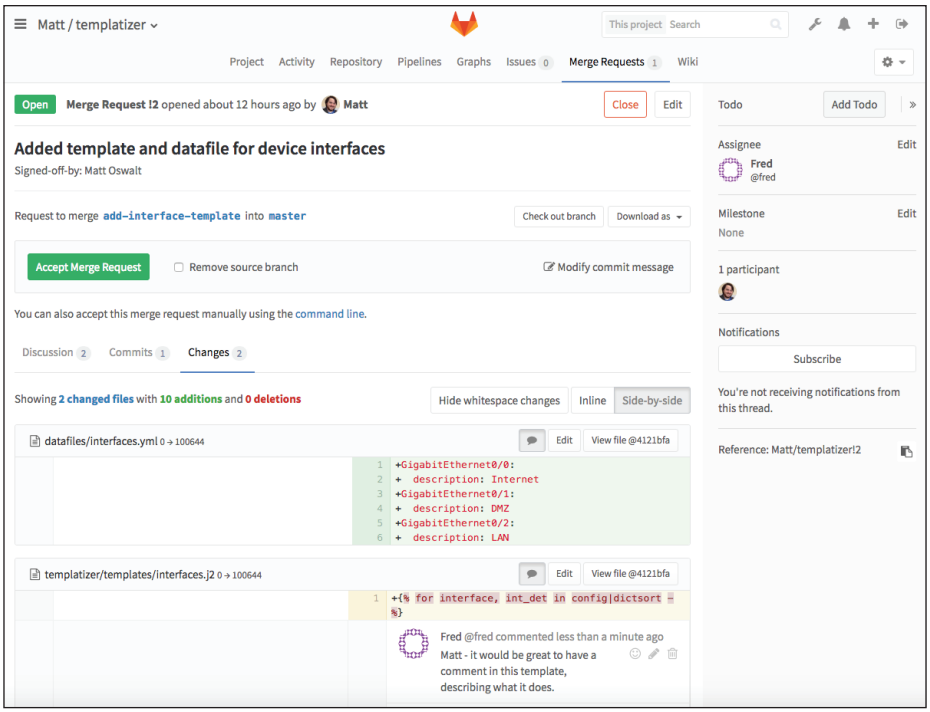


Рис. 10.9 ❖ Фред дает комментарии по запросу на объединение ветвей

Здесь можно видеть, что Фред оставил сообщение, в котором высказал свое мнение о необходимости добавления к предложенному шаблону комментария с кратким описанием функциональных возможностей этого шаблона. Достаточно часто предлагаемые изменения проходят цикл из нескольких итераций, прежде чем будут включены в основную ветвь, и почти все платформы предоставляют возможность для организации такого цикла. В системе GitLab нужно только добавить еще один коммит в текущую ветвь и передать его в сервис GitLab, после чего новый коммит будет автоматически внесен в рецензируемый запрос на объединение ветвей. Фред без каких-либо затруднений может увидеть дополнительные изменения, и если он согласен с тем, что обсуждае-

Tlgm: @it_boooks

мое изменение готово к включению в основную ветвь, то может дать положительную оценку.

На рис. 10.10 показано, как GitLab прослеживает весь этот поток событий, чтобы любой член группы мог наблюдать переходы между состояниями предложенного изменения.

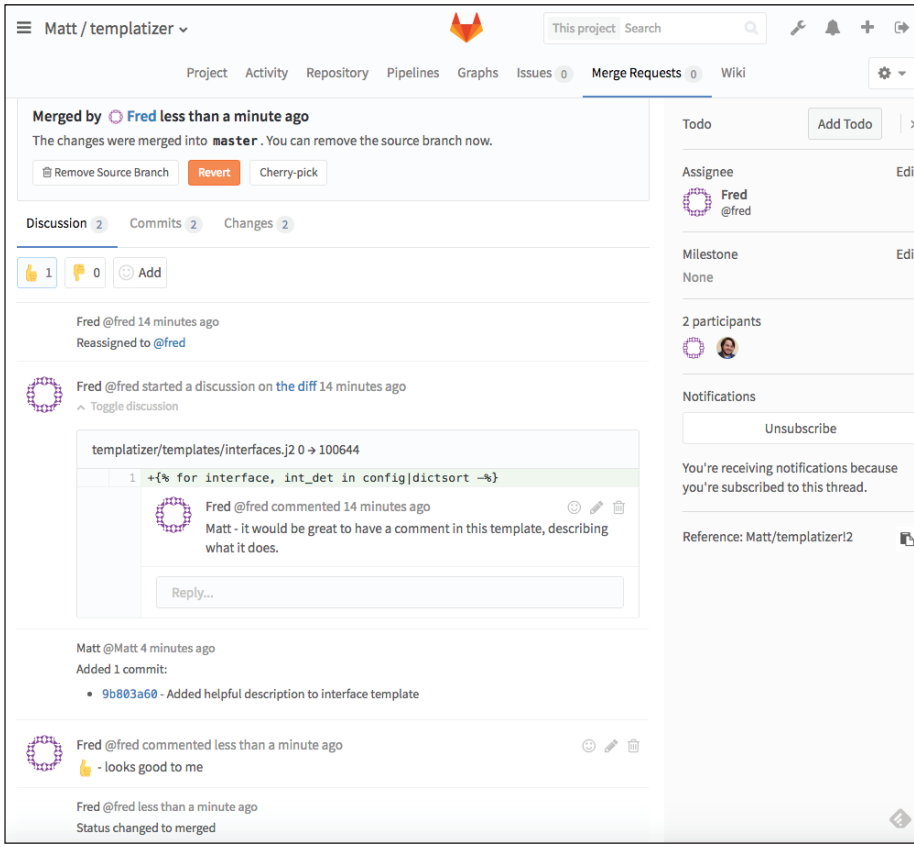


Рис. 10.10 ❖ Предложенное изменение принято, одобрено и включено в основную ветвь

Автоматизация сборки

Следующая весьма важная тема – автоматизация сборки (build automation). Этот термин появился при использовании инструментальных средств непрерывной интеграции как способа автоматической компиляции и/или установки (развертывания) ПО, для того чтобы приступить к его тестированию. Например, программа на языке C обязательно должна быть скомпилирована, чтобы появилась возможность ее запуска в среде тестирования.

Tlgm: @it_boooks

В рассматриваемом здесь конвейере не всегда необходима компиляция программ, но мы можем многократно использовать большинство задач, которые требуются разработчикам ПО для автоматического выполнения обработки каждого изменения, вносимого в репозиторий. Например, конвейер для проекта на языке Python может выполнять некоторый статический анализ кода, чтобы обеспечить соответствие правилам стиля PEP8. В контексте автоматизации сети изменения могут вноситься только в файлы YAML, но при этом можно выполнять аналогичные проверки для автоматизации некоторых простых объектов и операций, которыми нежелательно загружать людей-рецензентов. Например, проверка корректности содержимого файла YAML (то есть проверка правильности выравнивания и отступов в коде YAML).

Именно из-за этого автоматизация сборки становится столь важной. Еще перед привлечением к работе человека-рецензента можно автоматизировать множество процедур, чтобы рецензент действительно получил возможность давать полезные комментарии:

- статический анализ кода (проверка корректности синтаксиса и соответствия установленным правилам стиля);
- модульное тестирование (тесты отдельных модулей, синтаксический анализ файлов данных, шаблонов и т. д.);
- комплексное тестирование (не нарушает ли предлагаемое изменение существующую функциональность системы в целом).

После автоматического выполнения этих процедур сторонний рецензент может оставлять комментарии типа «это нужно сделать более удобным для чтения» вместо «добавь пробел здесь». По этой причине перечисленные выше автоматизированные процедуры обычно выполняются немедленно после передачи изменения в репозиторий (то есть сразу после создания запроса на объединение ветвей в предыдущем примере), а рецензент включается в работу после завершения всех проверок.

Такой процесс позволяет сэкономить время как для автора изменения, так и для рецензента, поскольку автор практически немедленно видит ответную реакцию, если его изменение вызывает какие-либо нарушения в работе системы, а рецензенту известно, что если изменение успешно прошло все предварительные тесты, то нет необходимости тратить время на простейшие комментарии, касающиеся элементарных вещей. Кроме того, при этом формируются воспроизводимые, более стабильные изменения в рабочем процессе автоматизации сети – при обнаружении ошибки ее корректировку можно добавить в комплект автоматизированных тестов для уверенности в том, что подобная ошибка не повторится снова.

Как уже было отмечено выше, существует ряд платформ, предоставляющих этот тип функциональности. Весьма часто выбирают Jenkins – сервер сборки с открытым исходным кодом и с возможностью интегрирования многочисленных интегрированных компонентов и функциональных возможностей. Но для наших примеров мы продолжим использование сервиса GitLab, так как он предлагает все необходимые функциональные возможности, которые объединены в одной программе. Кроме того, большинство автоматизируемых про-

Tlgm: @it_boooks

цедур в действительности можно реализовать с помощью скриптов в самом репозитории, сводя к минимуму зависимость от реально существующего сервера сборки и предоставляя максимум прозрачности для любого пользователя, работающего с этим репозиторием.

Предположим, что в недавно созданный файл *interfaces.yml* внесено небольшое изменение, и коллега Фред рецензирует его. Фред считает, что изменение правильное, дает оценку +1 и включает предложенное изменение в основную ветвь (см. рис. 10.11).

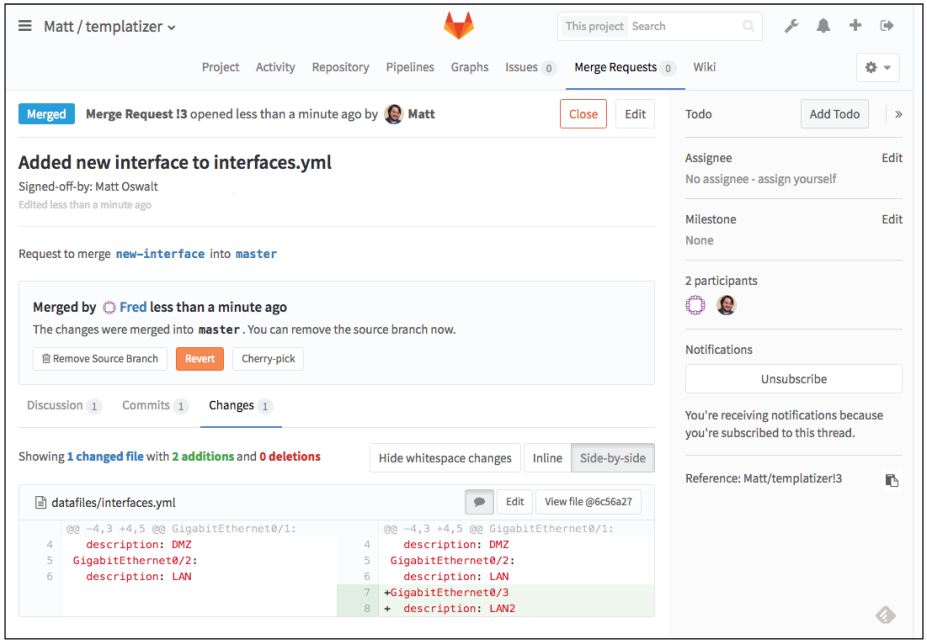


Рис. 10.11 ❖ Минимальное изменение в файле YAML

Тем не менее возникает проблема. Из-за этого изменения код YAML становится некорректным, что выясняется при попытке запуска программы Templatizer:

```
File "/Users/mierdin/Code/Python/templatizer/lib/python2.7/site-packages/yaml/scanner.py",
line 289,
  in stale_possible_simple_keys "could not found expected ':'", self.get_mark())
yaml.scanner.ScannerError: while scanning a simple key
  in "datafiles/interfaces.yml", line 7, column 1
could not found expected ':'
  in "datafiles/interfaces.yml", line 8, column 14
```

Было внесено самое минимальное изменение, но Фред тоже человек, и он не заметил опечатку, которую сделал Мэтт. При внесении более крупных исправ-

Tlgm: @it_boooks

лений, когда множество изменений вносится в несколько файлов, вероятность подобных ошибок и опечаток еще более возрастает.

С другой стороны, не составляет особого труда написать скрипт, отлавливающий ошибки такого типа, и обеспечить оперативную обратную связь с используемой системой сборки. Если это возможно, а конфигурация автоматизированной системы сборки позволяет выполнять такую проверку для всех последующих патчей (изменений), то в будущем мы сможем устранить эту проблему:

```
#!/usr/bin/env python

import os
import sys
import yaml

# YAML_DIR - место расположения каталога, в котором хранятся файлы YAML
YAML_DIR = "%s/../datafiles/" % os.path.dirname(os.path.abspath(__file__))

# Цикл прохода по всем файлам YAML с попыткой загрузки каждого файла
for filename in os.listdir(YAML_DIR):
    yaml_file = "%s%s" % (YAML_DIR, filename)

    if os.path.isfile(yaml_file) and ".yml" in yaml_file:
        try:
            with open(yaml_file) as yamlfile:
                configdata = yaml.load(yamlfile)

        # Если возникла проблема при импортировании YAML, можно вывести
        # сообщение об ошибке, затем завершить выполнение с ненулевым кодом ошибки
        # (который сработает в системе непрерывной интеграции как триггер,
        # сигнализирующий о критической неисправности)
        except Exception:
            print("%s failed YAML import" % yaml_file)
            sys.exit(1)

sys.exit(0)
```



Выше приведен предельно упрощенный пример. Существует несколько библиотек, которые могут обеспечить гораздо более подробную проверку корректности исходного кода. Рекомендуем библиотеку `pykwalify` для более тщательной валидации кода YAML (проверяется не только синтаксис, но и наличие ожидаемых значений).

После размещения приведенного выше скрипта в каталоге *tools* в нашем репозитории при работе с сервисом GitLab также необходимо изменить файл конфигурации конвейера непрерывной интеграции *.gitlab-ci.yml*:

```
test:
  script:
  - cd tools/ && python validate_yaml.py
```

После этого GitLab будет запускать данный скрипт при каждом внесении изменения. Скрипт проверки размещен в надлежащем месте, и можно узнать, что теперь увидит Фред, если Мэтт предложит еще одно изменение с некорректным кодом YAML (см. рис. 10.12):

Tlgm: @it_boooks

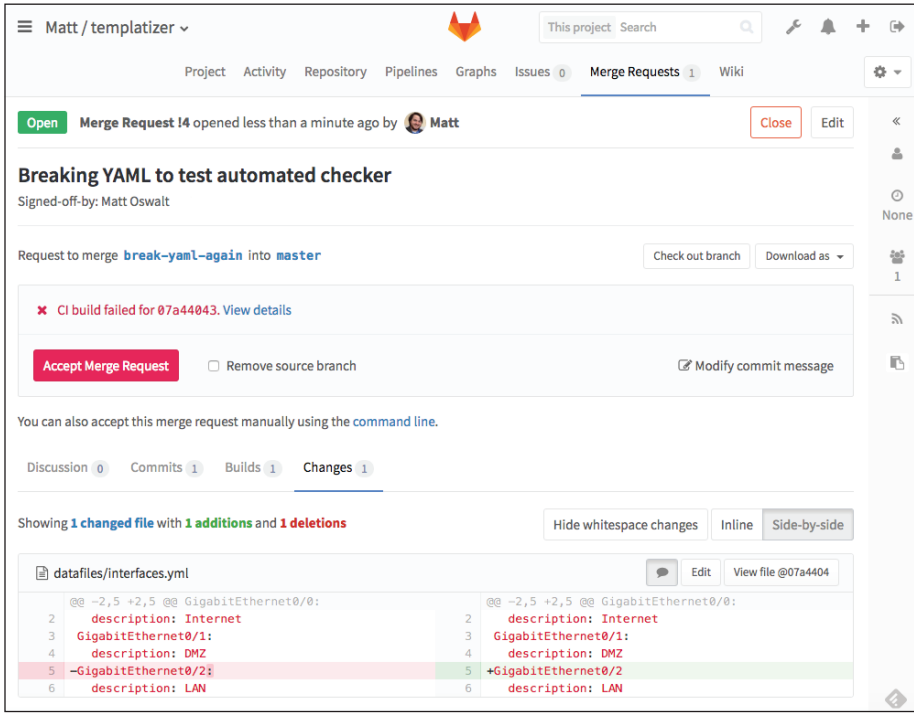


Рис. 10.12 ❖ Сборка в конвейере непрерывной интеграции завершилась с критической ошибкой из-за некорректного кода YAML

Мэтт и Фред получают возможность сразу же увидеть проблему, возникшую при автоматизированном тестировании. Кроме того, они могут посмотреть все подробности инцидента, включая полный журнал в консоли, где зафиксирована вся информация, выводимая скриптом проверки, с указанием конкретного файла, который стал источником проблемы (см. рис. 10.13).



Рис. 10.13 ❖ Вывод скрипта проверки корректности кода YAML

Tlgm: @it_boooks

Это всего лишь один из примеров, демонстрирующий богатые возможности, которые доступны при автоматизации процедур валидации и тестирования. Templatizer также является проектом, написанным на языке Python, поэтому можно применять некоторые средства из набора инструментов, доступного в этой экосистеме для выполнения некоторых специализированных для Python проверок и методик тестирования как части рассматриваемого здесь конвейера непрерывной интеграции. Например, Tox – широко распространенное инструментальное средство для выполнения всех типов автоматизированного тестирования в любом Python-проекте. Сообщество OpenStack использует Tox для эффективного упрощения процесса непрерывной интеграции, объединяя множество задач в небольшом списке команд:

```
test:
  script:
  - cd tools/ && python validate_yaml.py
  - tox -epep8  # Checks for PEP8 compliance with Python files
  - tox -epy27  # Runs unit tests
  - tox -ecover # Checks for unit test coverage
```

Еще раз отметим, что все эти команды обязательно должны быть выполнены без ошибок, чтобы процесс сборки прошел успешно. Когда рецензент получает запрос на объединение ветвей, в котором отмечено, что все проверки успешно пройдены, он уверен, что такой запрос действительно готов для рецензирования.

Эта часть конвейера чрезвычайно важна и представляет собой успешную методику сохранения эффективности рабочего потока, а также способствует устранению повторения прошлых ошибок. Ниже перечислены дополнительные концепции, которые могут оказаться полезными на этом этапе конвейера непрерывной интеграции, – рекомендуем проверить возможность их реализации в вашем процессе автоматизации сети:

- модульное тестирование любого кода (например, Python);
- комплексное тестирование, чтобы убедиться в том, что любой код способен взаимодействовать с другими проектами и API;
- валидация синтаксиса и стиля (не только исходного кода, но и форматов данных, таких как YAML).

Среда тестирования/разработки/перемещения данных

После этапа первоначального тестирования относительно простых характеристик, таких как синтаксис или стиль, обычно требуется выполнение тестирования «в более реальной среде» всех изменений, вносимых в репозиторий. Для программы Templatizer может потребоваться настоящая обработка реально существующих конфигураций с использованием шаблонов Jinja и файлов YAML на виртуальных устройствах, имитирующих подлинные рабочие устройства, которые в конечном итоге являются целевыми устройствами в нашем проекте. Для выполнения этой задачи существует несколько способов, и мы рассмотрим некоторые из них.

Tlgm: @it_boooks

Если необходимо выполнить относительно небольшой объем тестирования на обычном ноутбуке, то интересным решением является Vagrant. Vagrant – это инструментальное средство, разработанное компанией HashiCorp и предназначенное для упрощенного управления виртуальными машинами. Конфигурация этой среды содержится в файле Vagrantfile. Это обычный текстовый файл, который можно поместить в репозиторий Git. Он вполне пригоден для гарантии того, что каждый пользователь, работающий в репозитории, может инициализировать одинаковые виртуальные среды. Виртуальные образы также могут загружаться из предварительно определенной локации, так что пользователям необязательно компоновать и приводить в соответствие (синхронизировать) собственные образы. При правильной конфигурации пользователю необходимо только выполнить команду `vagrant up` в командной оболочке, и экземпляр среды будет создан автоматически.

Такой подход очень удобен для разработчиков ПО (Vagrantfile можно сконфигурировать для развертывания одинаковых сред разработки для всех программистов, работающих с единой кодовой базой), но сетевые инженеры также могут применять Vagrant для развертывания сетевых топологий с использованием устройств от многих различных производителей в «лабораторных» (испытательных) средах. Например, Juniper предоставляет свободный доступ к своим образам Junos, которые можно использовать вместе с Vagrant.

Ниже приведен пример разработки Vagrantfile, в котором формируется простая топология из трех сетевых узлов:

```
Vagrant.configure(2) do |config|

  # ПРИМЕЧАНИЕ: к моменту выпуска этой книги используемый в примере образ станет устаревшим.
  # Обратитесь к документации Juniper, чтобы получить более новый образ.
  config.vm.box = "juniper/ffp-12.1X47-D15.4-packetmode"
  config.vm.box_version = "0.2.0"

  config.vm.define "vsrx01" do |vsrx01|
    vsrx01.vm.host_name = "vsrx01"
    vsrx01.vm.network "private_network",
                      ip: "192.168.12.11",
                      virtualbox__intnet: "01-to-02"
    vsrx01.vm.network "private_network",
                      ip: "192.168.31.11",
                      virtualbox__intnet: "03-to-01"
  end

  config.vm.define "vsrx02" do |vsrx02|
    vsrx02.vm.host_name = "vsrx02"
    vsrx02.vm.network "private_network",
                      ip: "192.168.23.12",
                      virtualbox__intnet: "02-to-03"
    vsrx02.vm.network "private_network",
                      ip: "192.168.12.12",
                      virtualbox__intnet: "01-to-02"
  end
```

Tlgm: @it_boooks


```
  config.vm.define "vsrx03" do |vsrx03|
    vsrx03.vm.host_name = "vsrx03"
    vsrx03.vm.network "private_network",
                      ip: "192.168.31.13",
                      virtualbox__intnet: "03-to-01"
    vsrx03.vm.network "private_network",
                      ip: "192.168.23.13",
                      virtualbox__intnet: "02-to-03"
  end
end
```

В каталоге, где размещен этот Vagrantfile, необходимо просто выполнить команду `vagrant up`, после чего будет загружен образ Junos, инициализируются три виртуальных маршрутизатора Junos и устанавливаются соединения между ними в соответствии с описанием. Можно сразу же зарегистрироваться на любом из этих устройств без каких-либо затруднений:

```
~$ vagrant up vsrx01
Bringing machine 'vsrx01' up with 'virtualbox' provider...
(Инициализация виртуальной машины 'vsrx01' от провайдера 'virtualbox'...)
==> vsrx01: Checking if box 'juniper/ffp-12.1X47-D15.4-packetmode' is up to date...
           (Проверка актуальности образа 'juniper/ffp-12.1X47-D15.4-packetmode'...)
==> vsrx01: Clearing any previously set forwarded ports...
           (Удаление всех ранее установленных перенаправляемых портов...)
==> vsrx01: Clearing any previously set network interfaces...
           (Удаление всех ранее установленных сетевых интерфейсов...)
==> vsrx01: Preparing network interfaces based on configuration...
           (Подготовка сетевых интерфейсов на основе заданной конфигурации...)
    vsrx01: Adapter 1: nat
    vsrx01: Adapter 2: intnet
    vsrx01: Adapter 3: intnet
==> vsrx01: Forwarding ports...
           (Перенаправление портов...)
    vsrx01: 22 (guest) => 2222 (host) (adapter 1)
==> vsrx01: Booting VM...
           (Загрузка виртуальной машины...)
==> vsrx01: Waiting for machine to boot. This may take a few minutes...
           (Ожидание завершения загрузки виртуальной машины. Это может занять несколько
           минут...)
    vsrx01: SSH address: 127.0.0.1:2222
    vsrx01: SSH username: root
    vsrx01: SSH auth method: private key
==> vsrx01: Machine booted and ready!
           (Виртуальная машина загружена и готова к работе)
==> vsrx01: Checking for guest additions in VM...
           (Проверка предполагаемых дополнительных модулей для виртуальной машины...)
    vsrx01: No guest additions were detected on the base box for this VM! Guest
           (Не обнаружено дополнительных модулей для этой виртуальной машины.)
    vsrx01: additions are required for forwarded ports, shared folders, host only
           (Дополнительные модули требуются для перенаправления портов, для совместно
           используемых)
```

Tlgm: @it_boooks


```
    vsrx01: networking, and more. If SSH fails on this machine, please install
           (сетевых каталогов (папок), для создания хостов и т. п. Если на этом компьютере)
    vsrx01: the guest additions and repackage the box to continue.
           (не работает SSH, то установите необходимые дополнительные модули и повторно
            выполните упаковку образа для продолжения работы.)
    vsrx01:
    vsrx01: This is not an error message; everything may continue to work properly,
    vsrx01: in which case you may ignore this message.
           (Это не сообщение об ошибке: все компоненты могут продолжать корректную работу,
            и данное сообщение можно игнорировать.)
==> vsrx01: Setting hostname...
           (Установка имени хоста...)
==> vsrx01: Configuring and enabling network interfaces...
           (Конфигурирование и подключение сетевых интерфейсов...)
==> vsrx01: Machine already provisioned. Run `vagrant provision` or use the `--provision`
==> vsrx01: flag to force provisioning. Provisioners marked to run always will still run.
           (Виртуальная машина уже обеспечена всеми необходимыми ресурсами.
            Выполните команду `vagrant provision` или воспользуйтесь флагом `--provision`
            для явного указания использования ресурсов. Источники ресурсов помечены как
            активные в течение всего времени работы виртуальной машины.)

[вывод сокращен, поскольку выводимые данные аналогичны для vsrx02 и vsrx03...]

~$ vagrant ssh vsrx01
--- JUNOS 12.1X47-D15.4 built 2014-11-12 02:13:59 UTC
root@vsrx01% cli
root@vsrx01> show version
Hostname: vsrx01
Model: firefly-perimeter
JUNOS Software Release [12.1X47-D15.4]
```

Здесь показано, что на используемом компьютере достаточно ресурсов для начала работы описанной выше топологии, что требуемые образы доступны и существует несколько способов тестирования изменений, вносимых в сетевую среду. Это в какой-то мере беспрецедентный подход, так как ранее производители сетевого оборудования лишь после длительных задержек предоставляли свободный доступ к виртуальным образам своих платформ. Но в настоящее время все больше производителей начинают оперативно реагировать на потребности пользователей.

В применении к непрерывной интеграции и автоматизированному тестированию описанная концепция может оказаться весьма полезной для проверки корректности (валидации) вносимых изменений. Например, если вносится изменение в рассматриваемый ранее репозиторий Templatizer и это изменение необходимо протестировать перед развертыванием в эксплуатируемой рабочей среде, то можно привлечь к обработке один из таких шаблонов и автоматически развернуть вносимое в конфигурацию изменение в виртуальной топологии, предоставляемой Vagrant. Даже если необходимо протестировать что-либо вне рабочего компьютера (ноутбука), Vagrant остается чрезвычайно полезным инструментом для работы с сетевыми платформами.

Tlgm: @it_boooks

Кроме того, существует возможность работы виртуальной машины в общедоступной или частной облачной среде. Например, если ваша организация работает со средой OpenStack, то многие виртуальные сетевые устройства могут запускаться как виртуальные машины в OpenStack. Можно даже автоматизировать их развертывание с помощью шаблонов OpenStack Heat. В другом варианте такие компании, как Network to Code (http://networktocode.com/), предоставляют возможность автоматизации рабочего потока с помощью своей платформы On Demand Labs (https://labs.networktocode.com), предлагаемой сетевым инженерам для использования ресурсов общедоступной облачной среды при реализации подобных виртуальных топологий. Подобные методики очень удобны, если необходимо сделать одну топологию постоянно доступной для нескольких сетевых инженеров.



Дополнительная информация: Джейсон Эделман (Jason Edelman), один из авторов этой книги, является основателем компании Network to Code.

Многие из описанных вариантов можно автоматизировать с использованием большинства тех же инструментальных средств, которые применяются для автоматизации «настоящих устройств». Здесь важно, что постоянно используется одинаковый инструментальный комплект и в среде тестирования, и в реально эксплуатируемой среде, иначе тестирование стало бы бессмысленным. Например, если поставлена цель – использовать виртуальную среду для тестирования развертывания изменений в конфигурации с применением Ansible и создать виртуальную топологию, как можно более точно имитирующую реально эксплуатируемую рабочую среду, то необходимо организовать работу в одном и том же рабочем потоке Ansible и для среды тестирования, и для рабочей среды. Это существенно добавляет уверенности в том, что изменения, успешно прошедшие тестирование, будут корректно работать в эксплуатационной среде.

Среды тестирования являются одним из основных компонентов, которые заставляют организации всерьез задуматься о необходимости особой должности инженера, специально занимающегося сопровождением таких сред. Для обеспечения надлежащего уровня функционирования необходима тщательная поддержка тестовых сред, чтобы они не стали критическим «узким местом» в общем конвейере. Тестовые среды обязательно должны обеспечивать полноценную имитацию реальной сетевой среды.

Инструментальные средства развертывания

В предыдущей части главы обсуждалась важность полного понимания того, что именно развертывается в конвейере непрерывной интеграции/непрерывной поставки. Причина в том, что развертываемый объект оказывает существенное воздействие на выбор инструментальных средств, которые используются при реальном развертывании вносимых изменений.

Например, если вы пишете некоторый код на языке Python для автоматизации каких-либо задач в сетевой среде, то должны рассматривать этот процесс

Tlgm: @it_boooks

как полноценный проект по созданию ПО. Вне зависимости от объема, программный код, предназначенный для промышленной эксплуатации, всегда остается программным кодом. В небольшом скрипте могут содержаться ошибки, равно как и в крупном веб-приложении.

В дополнение к весьма важным процедурам тестирования и рецензирования коллегами, рассмотренными в предыдущих разделах, не менее важным может оказаться тщательное исследование механизмов поставки, которые начинают активно использовать разработчики ПО. Если организация пользуется облачными платформами, подобными OpenStack, то, вероятнее всего, ее сотрудникам предоставляется доступ к API, поддерживающим автоматическое развертывание вносимых изменений в конечной части конвейера непрерывной интеграции.

Кроме того, все более распространенной становится методика развертывания ПО в контейнерах Docker. Можно написать для конвейера непрерывной интеграции инструкции, в которых определяется автоматическое создание образа Docker в том случае, когда выполняется рецензирование и включение в основную ветвь очередного нового изменения. Такой образ может быть развернут в Docker Swarm или в кластере Kubernetes для реальной рабочей среды.

С другой стороны, иногда собственное специализированное программное обеспечение вообще не развертывается, то есть репозитории Git используются только для хранения таких объектов, как YAML-файлы или шаблоны Jinja. Это обычная практика для рабочего процесса автоматизации сети, в котором применяются инструменты управления конфигурацией, подобные Ansible, для принудительной передачи конфигураций устройств в сетевую инфраструктуру. Методы развертывания сетевых инженеров могут отличаться от методов развертывания разработчиков ПО, но в обоих случаях непрерывная интеграция играет важнейшую роль (см. рис. 10.14).

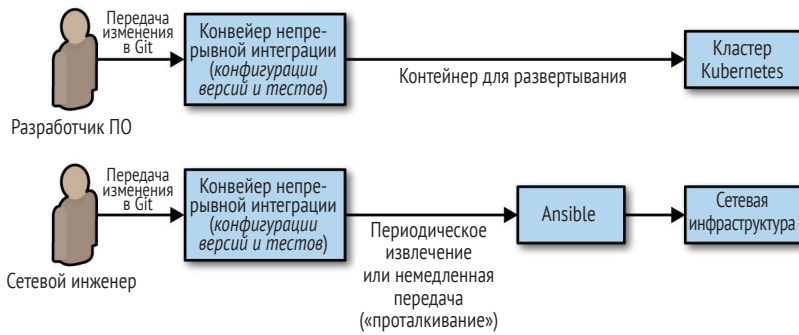


Рис. 10.14 ❖ Сравнение конвейеров непрерывной интеграции для разработки ПО и сопровождения сетевой среды

В нашем случае важно понимать, каким образом конфигурации устройств будут использоваться в рабочем режиме и как будут организованы откаты

Tlgm: @it_boooks

в предыдущее стабильное состояние. Это важная концепция не только для принятия решения о том, как Ansible будет в действительности работать в реальной эксплуатационной среде, но также для выбора методики формирования самих шаблонов конфигурации. Например, можно рассмотреть вариант выполнения комплекта сценариев Ansible для развертывания определенных шаблонов конфигурации для группы сетевых устройств при каждом вводе нового изменения в основную (master) ветвь репозитория, но какое воздействие это будет оказывать на всю конфигурацию в целом? Будет ли конфигурация всегда перезаписываться? Если будет, то не станет ли операция перезаписи потенциально опасной и даже критической частью конфигурации, что вовсе не предусматривалось?

Некоторые производители сетевого оборудования предоставляют инструментальные средства, помогающие решить эту проблему. Например, при передаче конфигурации на языке XML в устройство Junos можно воспользоваться флагом `operation` со значением `"replace"`, чтобы определить намерение заменить целый раздел конфигурации. В следующем примере показан шаблон Jinja для конфигурации устройства Junos, в котором используется этот флаг:

```
<configuration>
  <protocols>
      <bgp operation="replace">
          {% for groupname, grouplist in bgp.groups.iteritems() %}
          <group>
              <name>{{ groupname }}</name>
              <type>external</type>
              {% for neighbor in grouplist %}
              <neighbor>
                  <name>{{ neighbor.addr }}</name>
                  <peer-as>{{ neighbor.as }}</peer-as>
              </neighbor>
              {% endfor %}
          </group>
          {% endfor %}
      </bgp>
  </protocols>
</configuration>
```

К сожалению, не все производители позволяют выполнить такую операцию, но в данном конкретном случае предоставляется возможность просто заменять целые разделы конфигурации для каждого нового изменения («заплатки» – patch) в конвейере непрерывной интеграции таким способом, при котором можно быть уверенным: то, что должно быть (what it should be – WISB), вседа равнозначно тому, что есть на самом деле (what it really is – WIRI).

Это еще одна область деятельности, в которой невозможно предложить универсальное решение для любых ситуаций. Ответы на вопросы, возникающие при планировании развертывания, главным образом зависят от того, что именно вы развертываете и как часто выполняется этот процесс. Это наилуч-

Tlgm: @it_boooks

ший вариант самого первого этапа определения стратегии для автоматизации сети. Необходимо решить, требуются ли расходы на разработчиков и написание более специализированного программного обеспечения или достаточно использовать существующие инструментальные средства с открытым исходным кодом либо коммерческие инструменты для развертывания простых скриптов и шаблонов. Этим решением руководствуются при выборе наиболее подходящей модели развертывания.

Но важнее всего то, что развертывание никогда не должно выполняться, прежде чем будут реализованы вышеупомянутые концепции: рецензирование коллегами и автоматическое тестирование. Рабочий процесс автоматизации сети, в котором приоритет не отдан качеству и стабильности, обречен на провал.

Инструментальные средства тестирования и автоматизация сети по методике разработки через тестирование

В предыдущей части главы большое внимание было уделено влиянию методики разработки через тестирование на процесс автоматизации сети. В соответствующем разделе рассматривалось, насколько важно это, по сравнению с обычной сетевой статистикой, которая используется сетевыми инженерами, и дополнительными инструментальными средствами и метриками для выявления полезного практического опыта пользователей и применения конкретных приложений. Эти метрики можно использовать и до и после внесения каждого изменения, чтобы правильно оценить работоспособность и корректность сетевой среды и ее конфигурации (см. рис. 10.15).

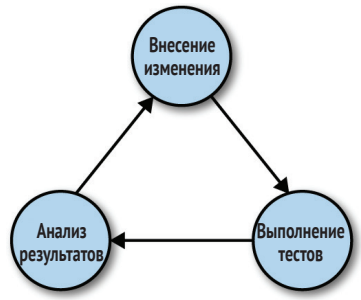


Рис. 10.15 ❖ Процесс непрерывного автоматизированного тестирования вносимых изменений

К сожалению, в течение нескольких десятилетий инструментальные средства, предназначенные для накопления полезного практического опыта применения программных приложений и для устранения проблем, почти не усовершенствовались и не развивались. В наше время устранение сетевых проблем обычно производится с помощью какого-либо представителя группы

Tlgm: @it_boooks

инструментальных средств, таких как ping, traceroute, iperf, а для любой платформы управления сетью возможен опрос по протоколу SNMP.

В большинстве случаев перечисленных выше инструментов недостаточно даже с точки зрения сетевого инженера, не говоря уже о том, что они способны предоставить лишь минимальные сведения о производительности приложений. Поэтому функциональность сетей мы считаем самым главным фактором. Но благодаря возникновению и развитию программ с открытым исходным кодом и предоставлению сервисов, подобных GitHub, которые существенно упрощают практическое применение (фактически – «потребление») ПО с открытым исходным кодом, ситуация начинает улучшаться.

Одной из областей, в которой назрели усовершенствования, особенно в сетевой инфраструктуре, является возможность сбора подробнейших телеметрических данных с использованием гибкой и масштабируемой практической методики. В настоящее время сетевые инженеры ограничены средствами, предоставляемыми протоколом SNMP, которому присущи некоторые недостатки. SNMP – это инструмент мониторинга, который не только ограничен самой сетевой инфраструктурой, но также способен предоставить лишь неполное подмножество данных, доступных на современных сетевых устройствах.

Основная проблема здесь состоит в том, что приходится игнорировать огромный объем действительно полезного контекста, доступного за пределами сетевой среды. Используя инструментальные рабочие среды, такие как Snap (https://github.com/intelsdi-x/snap) корпорации Intel, мы получаем возможность постоянно и осмысленно собирать телеметрические данные обо всех элементах инфраструктуры, относящиеся к сетевой среде, то есть о серверах приложений, о кластерах, о контейнерах и многих других объектах. При внесении в сетевую среду изменения, которое отрицательно влияет на один из таких объектов, можно сразу заметить это по доступным телеметрическим данным. Возможно, будет даже выполнен автоматический откат в предыдущее состояние, то есть отмена некорректных изменений на основе установленного порогового состояния.

Дополнительной практической методикой является активное тестирование инфраструктуры сети и приложений с использованием таких инструментальных средств, как ToDD (https://github.com/toddproject/todd), которые предоставляют механизм, обеспечивающий тестирование сетевой среды: ping, http, сканирование портов и тестирование пропускной способности полностью распределенными методами. ToDD также объединяет полученные данные в единый документ в формате JSON, что позволяет принимать решения на основе результатов тестирования вне зависимости от масштабов и объемов. Это очень важно при тестировании производительности на уровне приложений (не ограничиваясь только ping), а также при тестировании на уровне, сравнимом с режимом пиковой нагрузки в реальной эксплуатационной среде.



Проект ToDD был создан одним из авторов этой книги – Мэттом Осуолтом (Matt Oswalt).

Tlgm: @it_boooks

Инструментальные средства, подобные описанным выше, могут обеспечить дополнительную прозрачность во время обработки критического сбоя и восстановления нормального режима работы, например при имитации критического сбоя в центре обработки данных. Тестирование обработки критического сбоя является недооцениваемой операцией в сетевой инфраструктуре. Чаще всего трудно получить разрешение на проведение такого теста, но даже в тех редких случаях, когда удается добиться разрешения, еще труднее точно определить, как в действительности ведет себя сеть и работающие в ней приложения. Используя вышеупомянутые и другие инструментальные средства, можно собрать необходимые данные и сформировать основу, позволяющую определить «нормальную» производительность приложений. После этого при выполнении тестов после критических сбоев можно с большей уверенностью считать, что мы обладаем мощностями, достаточными для поддержания надлежащего уровня бизнес-деятельности.

Выше приведено всего лишь несколько примеров, подтверждающих основную мысль: ПО с открытым исходным кодом уже не является «элитным клубом» только для разработчиков программ. В настоящее время нет никаких причин, чтобы отказываться хотя бы от рассмотрения возможности применения инструментальных средств с открытым исходным кодом для заполнения огромных пробелов в существующих стратегиях и методиках мониторинга, особенно при явной недостаточности обеспечения прозрачности на уровне приложений при исследовании производительности сетевой среды.

Резюме

В наше время у любой организации есть неплохие возможности создать собственную группу или даже отдел разработки программного обеспечения. Установите связь с такими группами и узнайте подробности выполняемых ими процессов. Если группы разработки используют методику непрерывной интеграции, то весьма вероятно, что они позволят вам воспользоваться некоторыми из своих инструментов и практических методик для выполнения аналогичных задач в процессе автоматизации сети. Как уже было отмечено выше, релиз-инженер может оказать существенную помощь в управлении конвейером непрерывной интеграции.

В этой главе рассматривались разнообразные усовершенствования процесса (а также некоторые комплекты инструментов, способствующих улучшению рабочих процессов), но действительным стержнем всех этих преобразующих изменений является культура, обеспечивающая понимание затрат и преимуществ такого подхода. Эта культура более подробно описана в главе 11. Если вы не получаете средств от бизнеса для внедрения этих усовершенствований, то такой бизнес долго не протянет.

Кроме того, важно помнить, что неотъемлемой частью методики непрерывной интеграции/ непрерывной поставки является постоянный процесс обуче-

Tlgm: @it_boooks

ния. Необходимо постоянно поддерживать активное рабочее состояние и все время задавать себе вопрос: действительно ли текущая модель управления и мониторинга сетевой средой является достаточной. Требования к приложениям часто меняются, поэтому обычно ответ на этот вопрос звучит как «нет». Пострайтесь постоянно отслеживать новые и существующие программные приложения и быть в курсе событий в сообществах разработчиков ПО, чтобы своевременно узнавать о текущих требованиях и формировать конвейер, оперативно реагирующий на все изменения и усовершенствования.

Tlgm: @it_boooks

Глава 11

Формирование культуры автоматизации сети

Сетевая отрасль главным образом регулируется и управляется производимыми сетевыми устройствами и прочей продукцией, возможно, даже в большей степени, чем любая другая технологическая дисциплина. Чрезвычайно редко мы слышим о новых «революционных» процессах в ИТ или истории о том, как некая организация победила всех конкурентов благодаря своей великолепной ИТ-команде. Практически всегда всеобщим вниманием пользуется новая аппаратура или программные продукты.

Но даже самая «революционная» продукция сама по себе не решает проблемы бизнеса. Появление технологий виртуализации платформы x86 – вероятно, один из самых крупных прорывов, которые когда-либо происходили в области ИТ, но даже 10 лет спустя, несмотря на этот прорыв, мы тратим недели и даже месяцы на обеспечение виртуальных машин ресурсами. Разумеется, проблемы не ограничиваются только используемой технологией. Возможно, необходимо изменить еще и культуру. Именно об этом идет речь в данной главе – почему высокая культура является важнейшим основополагающим элементом автоматизации сети и как можно достичь высокого уровня такой культуры.

Но не следует чрезмерно увлекаться вопросами культуры и считать, что это единственное средство решения всех проблем. В действительности необходим баланс между подбором правильных людей, правильных процессов и правильной технологии, чтобы добиться успеха. Изменение культуры, обсуждаемое в данной главе, полностью ориентировано на удовлетворение стремления качественно выполнить требуемую работу в команде единомышленников.

В этой главе вы не найдете сведений о том, сколько раз в день нужно поощрять инженеров своей команды или почему в компании так важен батут в спортзале, чтобы осчастливить инженеров. Эта глава предоставляет возможность по-другому взглянуть на сущность «культуры», и вы сами убедитесь в том, что в основе заложена одна очень простая идея: люди хотят работать в команде с другими людьми, которым небезразлично то, что они делают.

Tlgm: @it_boooks

Если вы занимаете должность, которая подразумевает стремление сохранить существующий устраивающий всех порядок вещей на протяжении всей карьеры, и не собираетесь вносить какие-либо глобальные изменения, то эта глава вряд ли будет полезной для вас. Но если вы взяли в руки нашу книгу и дочитали до последней главы (кстати, выражаем признательность за это), то, вероятнее всего, вы ищете способы усовершенствований, хотя бы небольших, и не вполне удовлетворены существующим положением дел. Надеемся, что вы хотите стать тем человеком, который внедряет полезные изменения в своей организации и стремится постоянно повышать уровень своих технических знаний и практических навыков.

Исходя из этих предпосылок, в текущей главе мы будем рассматривать три основные темы:

- организационная стратегия и гибкость;
- восприятие ситуаций критических сбоев;
- практические навыки и образование.

Организационная стратегия и гибкость

Это самая первая концепция, которую должна принять ваша рабочая группа и которая должна надлежащим образом поддерживаться внутри организации. Если группа неверно понимает эту концепцию или вообще не принимает ее, то все прочие компоненты системы, рассматриваемые в данной книге, просто рухнут под собственным весом.

Преобразование организации старого образца

ИТ-предприятие не осведомлено в полной мере о своих возможностях, позволяющих находиться на переднем крае отрасли. Обычно набор (стек) технологий в типичной ИТ-компании может отставать на 5, иногда даже на 10 лет, по сравнению с предприятиями, применяющими самые передовые технологии и следящими за их развитием. Это явление не всегда возникает стихийно, ведь «передний край отрасли» не случайно получил такое название. Но обычно имеется множество не слишком убедительных объяснений того, что ИТ-компания продолжает использовать устаревший набор технологий как «наследство» прошедших времен.

Процесс автоматизации постоянно встречается с этим препятствием. Часто наблюдается заинтересованность некоторой группы людей, которые хотели бы начать автоматизацию, но не могут убедить кого-либо в своей организации сделать хотя бы первые шаги в этом направлении. Или кто-то начинает внедрять первые, самые простые элементы автоматизации, но совершает ошибки и приводит в нерабочее состояние систему, приносящую доход (этот вариант еще хуже, так как создает «дурную репутацию» для процесса автоматизации).

К сожалению, на все возникающие при этом вопросы нет простых ответов. В каждой организации имеется свой «боевой опыт» и собственная, отличаю-

Tlgm: @it_boooks

щаяся от прочих, история. Тем не менее один потенциально полезный ответ можно найти в знаменитой книге «Как завоевывать друзей и оказывать влияние на людей» (How to Win Friends and Influence People). В этой книге Дэйл Карнеги (Dale Carnegie) передает крупицы мудрости, весьма полезные в повседневной жизни, но одна из мыслей, многократно повторяющаяся на протяжении всей книги, заключается в необходимости взгляда на вещи с точки зрения других людей. В свободном изложении эта мысль выглядит так:

«*Невозможно заставить кого-либо делать то, что он не хочет делать. Поэтому нужно поступить так, чтобы он захотел сделать это*».

Более подробно такой подход мы обсудим в следующем разделе. Но пока кратко отметим, что деловая (и руководящая) часть организации должна «захотеть» внедрения автоматизации или, для начала, внедрения внутриучрежденческих скриптов как первого этапа автоматизации. На этом этапе не должно быть каких-то «высоконаучных» проектов, которые высшее руководство будет считать главной ошибкой, если что-то пойдет не так, как предполагалось.

Даже если вы начинаете с абсолютно правильных действий, при вынесении стратегии автоматизации на суд руководства бизнес-деятельности организации будет возникать определенная степень сопротивления. У предлагаемых изменений всегда находятся противники. Было бы странно, если бы не нашлось людей, которые почувствовали себя, по меньшей мере, неуютно в такой ситуации. Предлагая изменения, важно помнить о следующем: автоматизация необходима не потому, что она «крута и прогрессивна сама по себе» (даже если это действительно так), а потому, что она дает ощутимые, вполне измеримые преимущества для бизнес-деятельности.

Другой весьма важный факт состоит в том, что процесс необходимо выполнять без излишней спешки, аккуратно формировать его с учетом многолетних привычек и пристрастий инженеров, а кроме того, сформулировать правильные ожидаемые результаты с самого начала процесса. Автоматизация не похожа на потерю веса в процессе приобщения к здоровому образу жизни. Каждый, кто сумел похудеть и сохранить его в норме, расскажет не о жестких диетах, а о том, что нужно есть правильные продукты, разумными порциями и постоянно упражняться физически. Быстрые достижения совсем не так важны, как формирование здорового образа жизни на длительном отрезке времени. Диеты с жесткими ограничениями могут дать некоторые быстрые положительные результаты, которые сразу заметны, но вряд ли они обеспечат успех в дальнейшем.

Автоматизация также представляет собой постепенный процесс. Если в организации все внимание сосредоточено на формировании комплексной группы «автоматизации» или «DevOps», то такая деятельность заранее обречена на провал. Автоматизация – это один из видов деятельности, в которых требуется тщательно «удобрять почву» впрок и обеспечивать постепенный «естественный рост». Именно по этой причине организации, в которых уже

Tlgm: @it_boooks

сформирован достаточно высокий уровень автоматизации, не обозначают этот процесс каким-то особенным термином. Для них это всего лишь «современные операции».



Некоторые организации добились успеха с помощью временной «виртуальной» группы, созданной из специалистов в различных областях ИТ, перед которыми была поставлена задача внедрения автоматизации в конкретной организации. Это может быть удачным решением на начальном этапе, однако нельзя упускать из вида тот факт, что конечной целью является усовершенствование деятельности (рабочих операций) всей организации в целом, а не создание отдельной группы, которая занимается исключительно автоматизацией, в то время как остальным сотрудникам организации нет до этого никакого дела.

Чтобы подвести итог, еще раз напомним, что не следует пытаться «выпить море» или дать строго формальные определения всем объектам, которые необходимо автоматизировать в следующем столетии. Нужно просто начать. Начать с малого, сначала автоматизировать простые вещи и операции, даже если для этого потребуется написать несколько скриптов и запускать их с помощью сервиса cron, или в первую очередь сосредоточиться на автоматизации процедур, устраняющих проблемы и восстанавливающих нормальный режим работы. После того как вы начнете автоматизацию с простых объектов, обнаружится множество задач, также требующих выполнения, но вы будете чувствовать себя более уверенно, продолжая действовать в этом направлении.

Важность поддержки со стороны руководства

Еще раз подчеркнем важность того факта, что автоматизация выполняется с тщательно проработанной и определенной целью и стратегией и с обеспечением обмена информацией между всеми службами организации. Необходимо сосредоточиться на главной цели – получение прибыли от бизнес-деятельности, – не важно, будет ли это увеличение времени непрерывной работы, усовершенствование системы обеспечения безопасности или просто более быстрая ответная реакция на изменения условий и требований в деловой сфере. Эти характеристики (метрики) уже должны отслеживаться, и если вы рассчитываете начать автоматизацию без них, то вы приняли неправильное решение. В первую очередь необходимо подумать о том, насколько хорошо организована связь применяемых технологий (как в течение короткого времени, так и долговременно используемых) с целями бизнес-деятельности.

После принятия правильного решения по этому вопросу вы увидите весьма ощутимые выгоды от этого вскоре после начала процесса автоматизации. Во-первых, любое дополнительное обсуждение процесса автоматизации с последующим голосованием будет легче завершить успешно. Чаще всего недовольство инженеров, сопротивляющихся началу автоматизации, вызывает недостаток имеющихся ресурсов. При правильной организации обмена информацией с бизнес-службами буде обеспечено полное понимание того, что

Tlgm: @it_boooks

затраты на небольшое дополнительное голосование (опрос) будут весьма незначительными, по сравнению с потенциально возможными простоями и потерями, возникающими из-за того, что рабочие процессы выполняются исключительно вручную или частично автоматизированы с помощью инструментов, в спешке написанных инженером, перегруженным повседневной работой.

Но единственной наиважнейшей причиной частого обстоятельного обсуждения вопросов, касающихся автоматизации, с бизнес-руководством организации является возникновение ситуации, в которой процесс протекает не так, как предполагалось (это неизбежно). Невозможно полностью отказаться от всех новых инструментальных средств и процессов, вам придется продолжать работу с ними, чтобы исправить ситуацию. Мы обсудим это немного позже в текущей главе, но одной из причин, по которой крупнейшие веб-компании, такие как Facebook и Google, публично обсуждают свои методики восстановления после критических сбоев, является их полная уверенность в том, что из подобных критических ситуаций извлечен полезный урок и они не повторятся в будущем. Каждая критическая ситуация является возможностью для роста и развития. Такой план, принятый с самого начала бизнес-деятельности, обеспечит уверенность в том, что любые критические ситуации можно не только изучать, но и предусматривать («планировать»).

В качестве наглядного примера можно привести широко известный критический сбой с последующим перерывом в работе сервиса GitLab (SaaS-версия программы, которую мы использовали в главе 10) в январе 2017 года. Сервис не просто вышел из строя, для его восстановления потребовалось 18 часов из-за последовательности взаимосвязанных, ранее не выявленных неисправностей в процедурах резервного копирования. GitLab не закрылся от внешнего мира, чтобы в частном порядке выяснить причины аварии, а опубликовал в Google Doc общедоступный отчет обо всем, что им было известно на тот момент, так что пользователи могли узнать о том, что происходит в действительности. Сотрудники GitLab даже показывали «в прямом эфире» свою работу по возвращению сервиса в режим онлайн. После того как сервис снова стал доступным, GitLab не только представил в блоге подробнейшее описание причин и предпосылок критического сбоя и восстановительных операций, но, кроме того, рассмотрел меры, принятые для того, чтобы подобная ситуация не повторилась в будущем. Потребовалось немало времени, чтобы убедить пользователей в том, что GitLab весьма серьезно заботился (и заботится) о репутации своего сервиса и стремится извлечь полезные уроки из совершенных ошибок.

Завершая тему, следует отметить, что критические ситуации и сбои будут возникать неизбежно, поэтому для процесса автоматизации обязательным требованием является принятие правильного архитектурного проектного решения. Достижение договоренности о прозрачном и частом обмене информацией с бизнес-руководством предприятия поможет получить в свое распоряжение время и инструменты, необходимые для создания надлежащей основы для процесса автоматизации.

Tlgm: @it_boooks

Купить или создать самостоятельно

В наше время, когда почти все ИТ-сообщество вовлечено в «движение за открытый исходный код», легко поддаться влиянию этой популярнейшей инициативы. «Прекращайте покупать все подряд у производителей и поставщиков, создавайте все необходимое сами с помощью программного обеспечения с открытым исходным кодом» – так звучит своеобразный «лозунг» сегодня. К сожалению, эта декларация не соответствует действительности.

Несмотря на факты, внушаемые нам некоторыми аналитиками, крупные организации, такие как Facebook или Google, достижения которых в области автоматизации общеизвестны, не создают все свои инструменты и сервисы с нуля. В каждом компоненте своего стека технологий они находят компромиссные решения относительно того, что выгоднее с экономической точки зрения: «создать самостоятельно» или «купить готовый продукт». Например, эти компании вполне могут создать собственными силами серверы для своих огромных центров обработки данных, но не станут заниматься формированием каждого отдельного компонента из простейших элементов. Они продолжают покупать некоторые продукты, поскольку принято решение внедрять их для усовершенствования и расширения стека технологий. В некоторых случаях обнаруживается, что вполне приемлемым является готовое решение от компании Cisco для поддержки беспроводных соединений.

Здесь основная идея заключается в том, что каждый должен принимать собственные решения. Вероятнее всего, ваша организация не так велика, как Facebook или Google, поэтому вряд ли вы получите такую же пользу от создания собственных серверов, как эти компании, тем не менее есть возможность извлечь некоторые дополнительные, более мелкие преимущества из подобного подхода.

В корпоративной сфере ИТ существует проверенное практическое правило: покупная продукция может удовлетворить до 80 % функциональных потребностей организации. Производители не могут снабжать своей продукцией абсолютно каждое предприятие (как бы они ни старались), поэтому они вынуждены передавать собственные инженерные знания всем своим постоянным клиентам и поставлять продукцию, которая в достаточной мере подходит для немедленного решения текущих конкретных задач. Это означает, что при использовании стека технологий, приобретаемого таким способом, около 20 % ваших насущных конкретных задач остаются нерешенными. Обычно в корпоративной сфере ИТ просто принимают такое положение вещей как есть, но это неправильно.

Даже написание простейших скриптов может помочь в заполнении этого 20-процентного пробела. Например, имеется беспроводной контроллер, который не генерирует отчеты, необходимые для работы. Вместо того чтобы ждать месяцы/годы, пока производитель этого устройства внесет изменения в пользовательский интерфейс (а этого может и не произойти), узнайте, не прилагается ли к контроллеру API. При наличии API появится возможность написать

Tlgm: @it_boooks

скрипт на языке Python для извлечения требуемых данных, а с помощью какой-либо графической библиотеки можно сгенерировать удобное наглядное представление этих данных. Но это вовсе не означает, что вы сразу становитесь разработчиком ПО, это просто приобретение знаний, достаточных для написания скриптов, с помощью которых вы избавляетесь от ненужного многолетнего ожидания, пока производитель не отреагирует на запрос о дополнительных функциональных возможностях.

Оба аспекта парадигмы «купить или создать самостоятельно» обладают собственными характерными свойствами (см. рис. 11.1). Организации, выбирающие коммерческое, готовое к эксплуатации решение в одной из областей своей деятельности, склонны полагаться на внешние ресурсы для поддержки этого решения, в то время как стек технологий, создаваемый внутри предприятия, старается также сохранить внутреннюю модель поддержки. В последнем случае приобретается гораздо более солидный опыт, позволяющий выйти на уровень экспертов в этой области деятельности, самодостаточный в рамках организации.

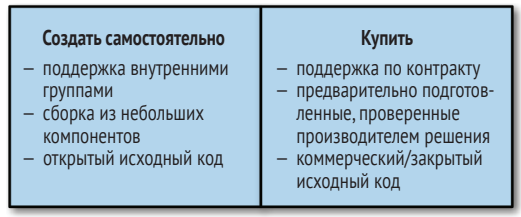


Рис. 11.1 ❖ Купить или создать самостоятельно

Но бинарный выбор (только одно из двух) не подходит для принятия решения о применении той или иной технологии. Более правильным является взгляд на парадигму «купить или создать самостоятельно» – как, впрочем, и на многие другие концепции, рассматриваемые в текущей главе, – как на своеобразный «спектр», состоящий из множества возможных решений. Ни одна организация не создает все с нуля. Каждая группа технологического обеспечения непременно должна четко определить правильный комплект методик для своей деятельности. В действительности любая организация использует сочетание обеих стратегий, рассматриваемых в текущем разделе.

Восприятие ситуаций критических сбоев

Нелегко сохранить правильное отношение к высказываниям крупных вебкомпаний о «мгновенной реакции на критический отказ» (failing fast) и «обработке критических ситуаций» (embracing failure). Ведь это звучит немного нелепо, не так ли? Обычно с критическими сбоями на уровне инфраструктуры связаны серьезные финансовые потери, поэтому неудивительно, что эти проблемы всегда обращают на себя повышенное внимание и вызывают дискуссии.

Tlgm: @it_boooks

Но споры по этому поводу чаще всего возникают из-за неправильного понимания сущности этих эффектно звучащих выражений. Сущность «обработки критических ситуаций» вовсе не состоит в том, что критическая ситуация воспринимается положительно. Тем не менее каким бы плохим ни был критический сбой, повторный (аналогичный) критический сбой еще хуже. Основная концепция, заложенная в основу «мгновенной реакции на критический отказ», состоит в недопущении повторного возникновения аналогичного критического сбоя. Поэтому необходимо предполагать, что критический сбой произойдет (потому что рано или поздно это случится), и иметь четко определенный план, который позволит извлечь полезный урок из критической ситуации. Может показаться, что некоторые крупные веб-компании слишком много внимания уделяют критическим сбоям и ситуациям. Причина в том, что при наличии рабочих процессов и сформированной корпоративной культуры эти компании обычно получают представление о критических ситуациях и сбоях, с которыми пока еще не сталкивались, поэтому получают возможность провести предварительную работу по решению новой проблемы.

Для прочих заинтересованных лиц концепция «обработки критических ситуаций» (embracing failure) отличается незначительно. Основная идея состоит в извлечении правильных уроков из любой критической ситуации, будь то отказ, вызванный ошибкой в используемой технологии, или чья-то неловкая опечатка в рабочем процессе автоматизации или в скрипте и последующая остановка работы всего центра обработки данных. Критические ситуации возникают и при автоматизации, и без нее, поэтому самое главное в этом случае – правильно воспринимать критическую ситуацию и составить план, в соответствии с которым организация должна реагировать на нее.



Именно поэтому так важно автоматическое тестирование. Правильная организация этого процесса означает, что тестирование является обязательным этапом при внесении любых изменений – это неотъемлемая часть процесса ввода изменений в эксплуатацию. Автоматизированные тесты фактически представляют собой переведенную на машинный язык версию уроков, извлеченных из прошлых критических ситуаций. Автоматизированное тестирование подробно рассматривалось в главе 10.

В предыдущем разделе подчеркивалась важность достижения полного взаимопонимания с бизнес-руководством организации, и это действительно очень важно. Критические ситуации являются неотъемлемой частью ИТ-деятельности вне зависимости от того, применяется автоматизация или нет. Достижение взаимопонимания с бизнес-руководством может превратить бурные споры с обвинениями типа «ваши скрипты совершенно вышли из-под контроля» в спокойное обсуждение полезных уроков, извлеченных из критической ситуации, и принятие мер по предотвращению повторного возникновения аналогичной ситуации. Сохраняйте «результаты вскрытия» и точно выясните, где именно возникла проблема, предъявите эти данные при обсуждении, чтобы перевести дискуссию в аналитическое русло, а не выслушивать поток обвинений. Критическая ситуация не всегда является свидетельством

Tlgm: @it_boooks

ваших неправильных действий, она может быть признаком того, что используемый стек технологий или набор практических навыков совершенствуется, а вы приобретаете дополнительный опыт, пусть даже и неприятный. Включите сценарии «что, если» в процесс обсуждения архитектурных решений, а также в процесс согласования ресурсов и целей с бизнес-руководством. Планируйте возникновение критических ситуаций при всех выполняемых вами действиях, и для вас не станет неожиданностью любая ситуация, в которой необходимо оперативно восстановить работоспособность центра управления сетью (network operations center – NOC).

Кроме того, критическая ситуация часто становится причиной прекращения автоматизации и возврата предприятия к режиму ручного управления процессами. Это может произойти практически незаметно. Когда процесс начинает протекать не так, как предполагалось, особенно при автоматизации сети, возникает искушение потихоньку отказаться от автоматизации и напрямую управлять ключевыми узлами инфраструктуры, как раньше. В зависимости от времени существования и степени развития процесса автоматизации такой вариант может оказаться исключительно единственным способом решения возникшей проблемы, так что это не всегда плохо.

Но настоящим пробным камнем «здорового процесса автоматизации» в любой организации является то, что именно делает организация для автоматизации после критического сбоя. Самые прогрессивные в этом плане организации немедленно начинают работу по усовершенствованию процесса автоматизации, с тем чтобы подобная критическая ситуация не возникла повторно. Из предыдущих примеров, в частности из описания ликвидации отказа сервиса GitLab, необходимо сделать следующий вывод: те, кто начинает активно действовать немедленно после устранения критической ситуации (сбоя), обеспечивают гарантию в том, что подобная ситуация никогда не повторится в будущем. Группы разработки ПО часто используют такой подход: при обнаружении ошибки в программе ошибка исправляется, но при этом создается модульный тест для воспроизведения условий и характеристик, которые стали причиной этой ошибки. Это позволяет обрести уверенность в том, что подобная ошибка не повторится в будущем (такие ошибки называют регрессивными).

Критические ситуации возникают неизбежно. Изучайте их, делайте выводы, извлекайте уроки и используйте процесс, который поможет избежать повторения таких ошибок.

Практические навыки и обучение

Владение необходимыми практическими навыками и знаниями всегда было важным в области ИТ, особенно если вы хотите чем-то отличаться от многочисленных коллег. Это становится еще более важным с учетом постоянно увеличивающейся скорости изменений в ИТ, поскольку вы можете в какой-то момент обнаружить, что ваш любимый стек технологий устарел и заменен чем-то новым.

Tlgm: @it_boooks

Если вы читаете эту книгу, вас можно поздравить – вы уже сделали первый большой шаг в правильном направлении. Главы книги написаны для профессионалов в области ИТ, которым необходимо «больше знаний», которым уже недостаточно привычной ориентированной на производителей оборудования затратной модели, применяемой на протяжении нескольких последних десятилетий.

В этом разделе рассматриваются возможности усовершенствования ИТ-образования и приобретения набора практических навыков, соответствующих следующему поколению технологий.

Изучайте неизвестное

Одним из самых частых ответов в беседе с коллегами или клиентами при обсуждении того, что можно сделать для начала процесса автоматизации, является фраза: «Я вообще не знал, что это возможно».

В самом деле, работа в области ИТ может порой создавать ситуацию «замкнутого пространства»: специалист постоянно слышит различные варианты одной и той же концепции, работает в основном над одними и теми же темами. Это один из самых опасных сценариев для любого технического специалиста, потому что подобная однобокость даже хуже, чем незнание другой технической дисциплины. При таком развитии событий вам совершенно неизвестно, чего вы не знаете (хотя, возможно, должны знать). Вряд ли вам потребуется использование языка Python для передачи команды выключателю в чулане, где хранятся инструменты для уборки и чистящие средства, потому что такие команды не приняты в реальном мире. Но если вы остаетесь в своем «замкнутом пространстве», то не получите никакого представления о том, что происходит во внешнем мире, и крайне замедлите свой рост как технический специалист.

К счастью, избежать такой ситуации не трудно. Почаще покидайте свою зону комфорта. Выходите во внешнюю среду, активно функционирующую за пределами вашего «уютного пространства». Существует великое множество технологий, которые могут вас заинтересовать, но вы даже не узнаете о них до тех пор, пока сами не прикажете себе заняться изучением новых, незнакомых областей.

Это имеет огромное значение для развития профессиональной карьеры, но, кроме того, важно для генерации новых идей в своей организации. Это преимущество не всегда ощутимо, но тем самым лишь подтверждается, как трудно получить одобрение и поддержку на совещаниях, особенно на совещаниях, где обсуждаются проблемы, выходящие за рамки вашей технической дисциплины. Если вы отвечаете за выбор основной темы совещания, то всегда помните о том, что правильный выбор основной темы является одним из наименее расточительных способов генерации свежих идей для вашей организации.

Короче говоря, покиньте свою «зону комфорта». В корпоративной ИТ-отрасли изменения не слишком велики и происходят не так часто, потому что

Tlgm: @it_boooks

внутрикорпоративная культура тесно связана и зависима от конкретных производителей и поставщиков ИТ-продукции, которые вполне обоснованно заинтересованы в сохранении текущего положения дел. Быстрые и частые изменения не соответствуют их модели коммерческой деятельности. Единственное, что можно сделать, чтобы изменить сложившуюся ситуацию, – остановить передачу идей и правил, навязываемых производителями и поставщиками. Вместо посещения крупной маркетинговой выставки, которую ваш поставщик растянул на целую неделю, возможно, лучше организовать менее масштабные мероприятия, например совещание локальной группы сетевых операторов. Или уделить внимание некоторым другим конференциям, на которых осуждаются темы вне пределов вашего набора знаний и навыков, например конференциям разработчиков ПО или конференциям по автоматизации.

Сосредоточьтесь на основных принципах

В любой технической дисциплине мы всегда встречаемся с новыми терминами, такими как «цифровое преобразование» (digital transformation) или «программно определяемый» (software defined), для описания новейшей технологии, выходящей на широкое рыночное пространство. Подобные термины создают ощущение того, что вы отстали в области бурно развивающихся технологий, и начинает казаться, что покупка самой последней версии продукта (физического или виртуального) вернет вас на передовые позиции.

В действительности большинство из нас не так уж сильно отстало за то время, в течение которого появлялись новые технологии. Особенно в корпоративной ИТ-отрасли, где отставание стека технологий может составлять 5–10 (возможно, даже больше) лет от того, что считается «передним краем» технологий, разрабатываемых в Кремниевой Долине. Тем не менее покупка новейшего разрекламированного продукта, предлагаемого вашим поставщиком, никогда не решала и не решит конкретную существующую проблему. Если такая ситуация когда-либо возникала, то она должна была разрешиться уже много лет назад. Действительная причина застоя в технологическом плане – недостаточные капиталовложения в людей и в обеспечение надлежащего уровня их технических навыков и знаний, а также зачастую слепое следование привычной линии поведения, диктуемой производителями и поставщиками ИТ-продукции.

В этой книге основное внимание уделено независимым от какого-либо конкретного производителя сетевого оборудования наборам практических навыков и знаний, процессам и культуре по одной простой причине – в действительности любая технология не способна изменить их в хоть сколько-нибудь значимой степени. Темпы развития и инвестиции увеличиваются и качественно улучшаются, а индустрия временами начинает развиваться в необычных направлениях, но это всегда маятниковое движение. Старые шаблоны и модели опять становятся новыми, а основополагающая технология, используемая на самом нижнем уровне, обычно не изменяется. Стек протоколов TCP/IP, знакомый и безоговорочно принятый всеми с самых первых дней появления

Tlgm: @it_boooks

комплекта сертификаций Cisco CCNA, и сейчас остается чрезвычайно важным для новейших программно определяемых сетей. Самые современные беспроводные устройства и протоколы продолжают сохранять в своей основе принцип использования радиочастот.

Это один из уроков, которые профессионалы в области создания сетевой инфраструктуры могут извлечь из деятельности разработчиков ПО. Вообще говоря, специалисты по сетевой инфраструктуре не столько «создают», сколько «приводят в действие», в то время как разработчики в большей степени привыкли мыслить как созидетели. Исходя из этого, можно сказать, что разработчики ПО обладают не каким-то всеобъемлющим навыком, а скорее набором микронавыков. Точно так же, как художник изучает такие навыки, как техника владения кистью и наука смешивания цветов, разработчики осваивают языки программирования, инструментальные средства, алгоритмы и приобретают знания об аппаратуре, прекрасно понимая, что в какой-то момент все это принесет пользу в новом крупном проекте.

В наше время обращает на себя особое внимание непрерывный рост важности программного обеспечения с открытым исходным кодом для всей ИТ-отрасли. При этом знание основных принципов создания систем и информационных технологий в целом никогда еще не было столь значимым. Изучайте ОС Linux. Осваивайте языки программирования. Эти базовые знания и навыки помогут прийти к более глубокому пониманию того, что нам уже удалось получить от использования информационных технологий всерьез и надолго. Новая ИТ-продукция появляется постоянно, но вся ее функциональность основана на аппаратном и программном обеспечении. Поэтому вне зависимости от того, стремитесь ли вы освоить новую техническую дисциплину или более глубоко изучить уже используемую вами технологию, сосредоточение на основных принципах является наилучшим способом, позволяющим идти в ногу с многочисленными, хотя и кажущимися мелкими изменениями в стеке ИТ-технологий на протяжении длительного времени.

Нужны ли сертификации?

Невозможно уйти от ответа на постоянно задаваемый вопрос: «Какова действительная ценность и значимость ИТ-сертификаций в эпоху автоматизации?» Это весьма сложный вопрос, особенно с учетом невозможности однозначного ответа на него для всех, поскольку каждый специалист занимает в ИТ-отрасли собственную карьерную позицию, отличающуюся от других.

Сертификации позволяют с определенной степенью уверенности предположить, что вы знаете материал, охватываемый конкретной темой сертификации. И несмотря на то что в настоящее время существуют проблемы, связанные с ИТ-сертификацией, никто не отрицает того, что это перспективный подход, заслуживающий внимания и дальнейшего исследования. Сертификация не обладает надлежащей степенью универсальности, позволяющей в полной мере определить ваши способности и возможности в данной об-

Tlgm: @it_boooks

ласти, но работодатель все же может узнать кое-что конкретное об этом. Без сертификаций вам придется начинать каждое интервью с нуля и доказывать потенциальному работодателю, что вы действительно знаете то, о чем говорите. Сертификации представляют собой неплохой способ обойти формальности начального этапа интервью, поэтому если вы новичок в ИТ-отрасли, то сертификации определенно являются хорошим средством, которое полезно иметь в своем арсенале.

Но сертификации влекут за собой и некоторые ограничения. Степень их значимости со временем снижается по мере того, как вы приобретаете новый практический опыт. Кроме того, часто сертификации в первую очередь идут на пользу производителям и поставщикам, поскольку они не могут в полном объеме дать вам все знания и навыки, которые вы хотели бы иметь. Возможно, в самом начале карьеры сертификации могут казаться полезными, но по мере приобретения практического опыта можно более глубоко изучить основные принципы и в меньшей степени зависеть от конкретных производителей и поставщиков продукции, демонстрируя работодателям свои истинные знания.

Если сосредоточиться на изучении основных принципов ИТ, то сертификации становятся в большей степени тактическим инструментом, нежели действенным способом получения профессионального образования. Сертификации эффективно помогают пройти через начальные этапы процесса приема на работу, и некоторые работодатели в обязательном порядке требуют наличия сертификаций. Тем не менее понимание основных принципов не только поможет успешно провести собеседование, но также обеспечит восхождение по карьерной лестнице, несмотря на все изменения, происходящие в сфере ИТ.

Может ли автоматизация лишить людей работы

Один из наиболее часто задаваемых вопросов относительно автоматизации звучит так: «А что будет с моей работой?» Широко распространено мнение о том, что автоматизация неизбежно означает сокращение рабочих мест. В конце концов, если машина способна выполнить некоторую работу, то кто согласится платить за ту же работу, выполняемую вручную?

Эта мысль основана на нескольких неправильных предпосылках. Первая заключается в том, что автоматизация каким-то образом происходит достаточно быстро, как смена дня и ночи, что в действительности абсолютно неверно. Автоматизация всегда является постепенным процессом, растянутым во времени на любом уровне. Сначала решаются простейшие задачи, затем происходит постепенный переход к более сложным, крупномасштабным задачам. Время от времени вы возвращаетесь на более ранние этапы и усовершенствуете то, что написали «в прошлом году».

Другая неправильная предпосылка – после того как автоматизация завершена, человеку делать нечего. Это также неверно не только по причине, описан-

Tlgm: @it_boooks

ной в предыдущем абзаце, но и потому, что автоматизация открывает новые функциональные возможности, о которых вы раньше даже не подозревали. Автоматизация действительно устраняет необходимость полной занятости человека, выполняющего определенную роль, но при этом ставит новые сложнейшие задачи, которые раньше просто невозможно было представить себе. Таким образом, люди должны переориентироваться на решение новых задач. В то время как какой-либо род деятельности может быть полностью автоматизирован, всегда появляются новые возможности, открывающиеся в процессе развития этого рода деятельности.

Поэтому вполне очевидно, что в организации, которая успешно провела процесс автоматизации, роли и обязанности изменятся. Не исчезнет необходимость в профессиональных, хорошо подготовленных сотрудниках, которым просто придется переориентироваться на решение новых сложных задач, возникших в процессе внедрения автоматизации.

Резюме

Авторы надеются, что эта глава помогла читателям усвоить весьма важную истину: инициативы, подобные DevOps, не связаны только с внедрением новых технологий или инструментальных средств. Нельзя также считать, что они лишь формируют новый процесс или даже новую культуру. DevOps объединяет все эти три аспекта. В любой организации необходимо применять методику и прививать сотрудникам системный образ мышления, аналогичный подходу к решению технических задач. DevOps уделяет большое внимание оптимизации системы с участием людей и усовершенствует обмен информацией так, чтобы все три вышеназванных «столпа» ИТ гармонично взаимодействовали. Правильно организованный процесс обмена информацией чрезвычайно важен. Успех невозможен без полного понимания того, как осуществляется бизнес-деятельность. Но вы также не достигнете успеха, если не сможете донести до коллег полную информацию о смысле того, что вы делаете.

Многие концепции, изложенные в этой главе, основаны на уверенности в правильности этой основной идеи. Сообщество часто склонно к решениям любых проблем в «бинарном стиле», то есть предпочитает однозначный выбор одной из двух противоположных точек зрения, но в действительности зачастую более практичным является «спектральный стиль» (то есть некоторый диапазон совместимых вариантов). Это в полной мере относится и к ИТ-сфере. То, что работает в одной организации, может быть совершенно неприменимо в другой. Вы сами должны выбрать наиболее прагматичный подход и целенаправленно решать поставленные перед вами конкретные задачи. Не следует полагаться на ИТ-аналитику и на опыт крупных веб-компаний в процессе принятия собственных стратегических технических решений.

В заключение отметим, что необходимый вам учебный курс невозможно разместить в одной главе (и даже в целой книге). Рекомендуется установить

Tlgm: @it_boooks

постоянную связь с другими сообществами специалистов, которые уже прошли этот путь, и у вас появится возможность учиться на их ошибках. Одним из таких полезнейших ресурсов для практического обучения является сообщество Slack, организованное инициативной группой «Network to Code» (можно оформить бесплатную подписку на сайте http://slack.networktocode.com/). Здесь вы найдете более 50 каналов по различным темам, связанным с инфраструктурой автоматизации и дополнительно подразделенным по производителям оборудования и по проектам с открытым исходным кодом. Это очень полезный источник разнообразной информации, особенно если вы только начинаете свою работу в области автоматизации сети.

Наконец, хочется пожелать всем читателям: активно занимайтесь автоматизацией – не становитесь объектом автоматизации сами.

Tlgm: @it_boooks

Приложение A

Профессиональное управление сетевой средой в ОС Linux

В главе 3 рассматривались некоторые основные концепции работы с сетью в ОС Linux. В этом приложении мы воспользуемся базовым материалом главы 3 в качестве основы для обсуждения нескольких расширенных концепций работы в сетевой среде и более сложных сетевых конфигураций для ОС Linux.

Основные темы приложения:

- использование интерфейсов macvlan;
- виртуальные машины в сетевой среде;
- работа с сетевыми пространствами имен;
- контейнеры Linux в сетевой среде;
- использование Open vSwitch (OVS).

По большинству из перечисленных тем можно написать отдельные книги. Поэтому при обсуждении этих тем не будет поставлена задача полного и глубокого изложения. Мы сосредоточимся на предоставлении объема информации, достаточного для понимания того, какое место занимают перечисленные темы в общей картине сетевой среды. Кроме того, будут изложены основные сведения о том, как установить, настроить и обеспечить управление этими сетевыми конфигурациями.

Начнем с рассмотрения практического использования интерфейсов macvlan.

Использование интерфейсов macvlan

Интерфейс macvlan представляет собой в некотором роде противоположность интерфейсу виртуальной локальной сети VLAN, который рассматривался в главе 3. Интерфейсы VLAN позволяют одному физическому интерфейсу вести обмен данными одновременно с несколькими виртуальными локальными сетями (VLAN; или широковещательными доменами – broadcast domains).

Tlgm: @it_boooks

Это можно интерпретировать как схему типа «несколько сетей на одном физическом интерфейсе». Противоположный подход представляют интерфейсы macvlan, которые позволяют создавать несколько логических интерфейсов в одном широковещательном домене, то есть это можно назвать схемой типа «одна сеть с несколькими логическими интерфейсами». Каждый логический интерфейс macvlan при этом будет иметь собственный адрес MAC (Media Access Control), и ему будет разрешено видеть только трафик, предназначенный для этого MAC-адреса. (Иначе говоря, один интерфейс macvlan не может перехватить трафик, предназначенный для другого интерфейса macvlan.)

Это может показаться немного непонятным на первый взгляд, поэтому в следующем разделе приведена пара примеров, демонстрирующих практическое применение функциональности интерфейсов macvlan.

Варианты практического использования интерфейсов macvlan

В настоящее время интерфейсы macvlan на практике применяются в двух конкретных вариантах:

- при объединении хостов и намерении сохранить в секрете MAC-адрес и IP-адрес каждого хоста можно воспроизвести интерфейсы для них как интерфейсы macvlan на вновь формируемых хостах. Это позволит сервисам продолжить работу без каких-либо изменений, даже если сервисы теперь запускаются на другом хосте;
- возможно использование интерфейсов macvlan вместо обычного шлюза (bridge) Linux в тех случаях, когда не требуется полная функциональность шлюза. Ниже рассматриваются примеры реализации этого варианта: один в следующем подразделе, второй – немного позже в текущем приложении.

Теперь, когда известно, в каких случаях возможно применение интерфейсов macvlan, можно перейти к более глубокому рассмотрению технических подробностей работы с этими интерфейсами.

Создание, конфигурирование и удаление интерфейсов macvlan

Для создания интерфейса macvlan используется уже знакомая нам команда `ip`, точнее, команда `ip link`. Общий синтаксис полной команды добавления интерфейса macvlan выглядит так:

```
ip link add link parent-device macvlan-device type macvlan
```

Рассмотрим эту команду более подробно:

- *`parent-device`* – физический интерфейс, с которым должен быть связан новый интерфейс macvlan. Иногда его обозначают термином «устройство более низкого уровня» (lower device);
- *`macvlan-device`* – имя, присваиваемое новому логическому интерфейсу macvlan. В отличие от интерфейсов VLAN, не существует каких-либо установленных соглашений об именовании интерфейсов macvlan.

Tlgm: @it_boooks

Предположим, что необходимо создать интерфейс macvlan в системе CentOS. Новый логический интерфейс должен быть связан с физическим интерфейсом ens33. Для решения этой задачи выполняется следующая команда:

```
[vagrant@centos ~]$ ip link add link ens33 macvlan0 type macvlan
[vagrant@centos ~]$
```

Если необходимо создать интерфейс macvlan с конкретным MAC-адресом (в предыдущей команде MAC-адрес генерировался автоматически), то необходимо вставить опцию `address` *`требуемый-MAC-адрес`* между именем нового устройства macvlan и завершающей частью команды `type macvlan`.

После создания интерфейса его действительное существование можно проверить с помощью команды `ip link list` и ключа `-d`, который выводит дополнительную информацию об интерфейсах VLAN, в том числе и об интерфейсах macvlan:

```
[vagrant@centos ~]$ ip -d link list macvlan0
6: macvlan0@ens33: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether b6:73:dc:60:a3:10 brd ff:ff:ff:ff:ff:ff promiscuity 0
    macvlan  mode vepa
```

Обратите внимание на последнюю строку `macvlan mode vepa`, она показывает текущий режим работы устройства macvlan. Режим `vepa` означает, что на Linux-хосте подразумевается наличие коммутатора трафика в восходящем направлении (upstream switch) с поддержкой протокола 802.1Qbg, в котором на момент написания данной книги имелись определенные ограничения. Также доступны и другие режимы: кроме `vepa`, можно использовать `bridge`, `private` и `passthru`. С высокой вероятностью в большинстве случаев наиболее подходящим будет режим `bridge`, поэтому его можно установить непосредственно при создании нового интерфейса или позже.

Для установки требуемого режима при создании интерфейса выполняется команда:

```
[vagrant@centos ~]$ ip link add link ens33 macvlan0 type macvlan mode bridge
[vagrant@centos ~]$
```

Если необходимо изменить режим после создания интерфейса, то используется команда ip link set:

```
[vagrant@centos ~]$ ip link set macvlan0 type macvlan mode bridge
[vagrant@centos ~]$ ip -d link list macvlan0
6: macvlan0@ens33: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether b6:73:dc:60:a3:10 brd ff:ff:ff:ff:ff:ff promiscuity 0
    macvlan  mode bridge
[vagrant@centos ~]$
```

Как и для (почти) всех прочих типов интерфейсов, необходимо активизировать созданный интерфейс (установить для него состояние `up`) и присвоить ему IP-адрес, чтобы полностью подготовить его к работе:

Tlgm: @it_boooks


```
[vagrant@centos ~]$ ip link set macvlan0 up
[vagrant@centos ~]$ ip addr add 192.168.100.112/24 dev macvlan0
[vagrant@centos ~]$
```

Чтобы удалить ненужный интерфейс macvlan, сначала необходимо отключить его командой `ip link set`, затем удалить командой `ip link delete`:

```
[vagrant@centos ~]$ ip link set macvlan0 down
[vagrant@centos ~]$ ip link delete macvlan0
[vagrant@centos ~]$
```

Как сделать различные конфигурации интерфейсов macvlan постоянно существующими? По умолчанию системы RHEL/Fedora/CentOS (на момент написания данной книги) не содержали каких-либо средств, с помощью которых можно было бы воспользоваться файлами конфигурации для каждого отдельного интерфейса в каталоге */etc/sysconig/network-scripts* для создания постоянно существующих конфигураций интерфейсов macvlan. Существует несколько способов обхода этого ограничения, например один из возможных вариантов можно найти в репозитории GitHub (https://github.com/larsks/initscripts-macvlan).

В системах Debian/Ubuntu проблема решается использованием функциональности `pre-up` в фрагментах сетевой конфигурации для запуска дополнительных команд перед созданием и активизацией сетевого интерфейса. В приведенном ниже примере создается постоянно существующая конфигурация интерфейса macvlan, связанного с физическим интерфейсом eth2:

```
auto macvlan0
iface macvlan0 inet static
  address 192.168.100.110/24
  pre-up ip link add link eth2 macvlan0 type macvlan
```

Виртуальные машины в сетевой среде

О возможностях работы в сетевой среде виртуальных машин, работающих под управлением гипервизора на основе ОС Linux, можно написать отдельную книгу, но в этом разделе мы ограничимся расмотрением двух конфигураций высокого уровня, позволяющих охватить большинство вариантов реализации, с которыми вы, вероятнее всего, будете часто встречаться в реальной практике. При этом из-за необходимости краткого изложения материала в этом разделе мы ограничимся рассмотрением только гипервизора KVM (отдавая ему предпочтение перед Xen и другими решениями на основе ОС Linux) и использованием в основном только API виртуализации Libvirt. Разумеется, существуют и другие гипервизоры, другие инструментальные средства и другие конфигурации, но, к сожалению, невозможно рассмотреть все возможные варианты и сочетания в рамках одного раздела.

Здесь мы будем рассматривать две сетевые конфигурации виртуальных машин:

- сетевые виртуальные машины, использующие шлюз Linux;
- сетевые виртуальные машины, использующие интерфейсы macvtap.

Tlgm: @it_boooks

Начнем с примера использования шлюза Linux.

Использование шлюза

Использование в качестве шлюза виртуальных машин в физической сети через один или несколько физических интерфейсов хоста Linux представляет собой весьма часто встречающийся вариант реализации шлюза Linux. В одном из примеров главы 3 был представлен первый, упрощенный вариант реализации применения шлюзов в ОС Linux. В этом разделе задача использования шлюзов рассматривается более подробно.

При использовании виртуальных машин для сетевого шлюза с применением гипервизора KVM и API Libvirt мы имеем дело с несколькими различными компонентами:

- шлюз Linux (разумеется);
- виртуальная сеть Libvirt, которая сообщает API Libvirt, какой именно шлюз Linux используется;
- интерфейс виртуальной сети;
- гостевой домен KVM (термин «домен» здесь используется для обозначения виртуальной машины, работающей под управлением KVM).

Необходимо объединить все эти компоненты для совместной работы.

Вообще говоря, в первую очередь необходимо воспользоваться Libvirt для создания виртуальной сети, определяя ее в формате языка XML. (Если вам незнаком язык XML, не стоит беспокоиться – изучите главу 5 данной книги.) Виртуальная сеть – это абстракция, используемая API Libvirt для обозначения (и описания) конкретной сетевой конфигурации более низкого уровня. Такой сетевой конфигурацией более низкого уровня может быть шлюз (как в нашем примере) или некоторая другая конфигурация (как мы увидим в следующем подразделе).

Libvirt использует XML для определения этих абстрактных объектов, в том числе и виртуальных сетей. Приведенный ниже код XML предназначен для создания виртуальной сети network-br0, которая обозначает шлюз Linux с именем br0. Отметим, что системный администратор должен создать br0 и установить связь между физическим интерфейсом и этим шлюзом, используя процедуры и команды, описанные в главе 3.

```
<network>
  <name>network-br0</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
</network>
```

Чтобы сообщить домену KVM о необходимости использования этой виртуальной сети, вы должны создать конфигурацию домена на языке XML, пример которой приведен ниже (здесь показана только часть XML-определения домена, относящаяся к сетевой среде, но в нее включены не все возможные опции и определения):

Tlgm: @it_boooks


```
<interface type="network">
  <source network="network-br0"/>
</interface>
```

В этом примере Libvirt оповещается (с помощью XML-определения гостевого домена) об установлении связи с виртуальной сетью Libvirt, именуемой network-br0. В свою очередь, виртуальная сеть Libvirt связана со шлюзом Linux br0. Преимущество использования абстрактной виртуальной сети состоит в том, что появляется возможность переключения на более низком уровне со шлюза br0 на шлюз br1 с помощью простого изменения определения виртуальной сети. При этом исключается необходимость каких-либо изменений для любых виртуальных машин, поскольку они связаны с одной виртуальной сетью.

После создания показанной выше конфигурации и активизации гостевого домена KVM Libvirt автоматически создает интерфейс виртуальной сети (устройство TAP) и подключает его к шлюзу с соответствующим определением виртуальной сети Libvirt (в нашем примере это шлюз br0). Гостевой домен получает в свое распоряжение сетевой интерфейс, который связывается гипервизором с устройством TAP. Таким образом, создается «цепочка» связей: гостевой интерфейс eth0 связывается с устройством TAP, которое связывается со шлюзом, для которого устанавливается связь с физическим интерфейсом и с сетью, к которой подключен последний.

API Libvirt автоматизирует почти все описанные выше действия и операции, которые не так-то просто проследить во время их выполнения. Тем не менее имеется возможность настройки всей «цепочки» вручную, чтобы наблюдать процесс объединения всех компонентов. Ниже мы продемонстрируем всю последовательность шагов, необходимых для использования виртуальной машины в качестве шлюза в сетевой среде. Ручная настройка не рекомендуется в реальной эксплуатационной среде, но как учебный пример может быть полезной для лучшего понимания того, что «происходит внутри» при использовании KVM и Libvirt.

В первую очередь необходимо создать шлюз Linux и подключить к нему физический интерфейс. Предположим, что к шлюзу подключается интерфейс eth1, тогда нужно выполнить следующие команды:

```
vagrant@trusty:~$ ip link add name br0 type bridge
vagrant@trusty:~$ ip link set br0 up
vagrant@trusty:~$ ip link set eth1 master br0
vagrant@trusty:~$ ip link set eth1 up
vagrant@trusty:~$
```



Отметим, что вне зависимости от того, подключаются ли виртуальные машины к шлюзу вручную или для этого используется Libvirt, в любом случае выполняются различные команды `ip` для создания шлюза Linux и подключения к нему одного или нескольких интерфейсов. Особо отметим, что интерфейсы, подключаемые к шлюзу, могут быть интерфейсами виртуальных сетей VLAN.

Tlgm: @it_boooks

Далее необходимо создать устройство TAP с помощью команды, которую мы до сих пор не использовали: `ip tuntap`. Общая форма этой команды: `ip tuntap add` *имя-устройства* `mode tap`. Если нужно задать имя tap0 для устройства TAP, связываемого с нашей виртуальной машиной, то выполняются следующие команды:

```
vagrant@trusty:~$ ip tuntap add tap0 mode tap
vagrant@trusty:~$ ip link set tap0 up
vagrant@trusty:~$ ip -d link list tap0
5: tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN
    mode DEFAULT group default qlen 500
    link/ether 7e:28:d5:99:ca:ab brd ff:ff:ff:ff:ff:ff promiscuity 0
    tun
vagrant@trusty:~$
```

Первая команда создает устройство TAP, вторая – разрешает установление соединения, третья – проверяет состояние устройства. В данных, выводимых третьей командой, сообщается, что интерфейс активизирован («разрешен»), но не установлено соединение с каким-либо устройством или объектом (характеристика `NO-CARRIER` в первой строке вывода).

После этого устройство TAP добавляется к существующему шлюзу, затем выполняется проверка правильности подключения с помощью команды `bridge link`:

```
vagrant@trusty:~$ ip link set tap0 master br0
vagrant@trusty:~$ bridge link list
3: eth1 state UP : <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br0 state
forwarding priority 32 cost 4
5: tap0 state DOWN : <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 master br0 state
disabled priority 32 cost 100
vagrant@trusty:~$
```

На заключительном этапе запускается виртуальная машина и подключается к устройству TAP. Здесь не будут рассматриваться все подробности используемых команд, поскольку они вряд ли потребуются в реальной практической деятельности (с большей вероятностью вы предпочтете воспользоваться командой `virsh` из библиотеки Libvirt). Отметим, что в командах, которые не умещаются на одной строке, при переносе используется символ обратного слеша, чтобы экранировать символ перехода на новую строку. В таком виде команды удобнее читать.

```
vagrant@trusty:~$ qemu-system-x86_64 -enable-kvm -hda cirros-01.qcow2 \
-net nic -net tap,ifname=tap0,script=no,downscript=no -vnc :1 &
[1] 866
vagrant@trusty:~$
```

Эта команда загружает домен KVM в фоновом режиме. После этого можно выполнить команду `ip -d link list tap0`, чтобы убедиться в том, что устройство TAP активно (отметим, что команда `bridge link list` также показывает, что устройство TAP подключено и активно):

Tlgm: @it_boooks


```
vagrant@trusty:~$ ip -d link list tap0
5: tap0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0
    state UP mode DEFAULT group default qlen 500
    link/ether 7e:28:d5:99:ca:ab brd ff:ff:ff:ff:ff:ff promiscuity 1
    tun
    bridge_slave
vagrant@trusty:~$
```

Если в этом сегменте сети работает сервер DHCP, с которым установлено соединение через интерфейс eth1 хоста KVM, то созданный только что гостевой домен должен получить IP-адрес и стать доступным для других систем (узлов) этой подсети.

Еще раз напомним, что совсем не обязательно выполнять все описанные выше шаги вручную, чтобы создать сетевой шлюз с использованием KVM и Libvirt. Здесь поэтапное выполнение команд вручную приведено для того, чтобы вы лучше поняли, что происходит на каждой стадии процесса. KVM и Libvirt автоматизируют большую часть этих этапов.

Кроме того, после подробного рассмотрения интерфейсов VLAN можно с уверенностью сказать, что эти интерфейсы вполне пригодны для использования в сетевом шлюзе. Один из возможных способов соединения через шлюз различных виртуальных машин, управляемых одним гипервизором Linux, в разных виртуальных сетях VLAN – создание шлюза для каждой виртуальной сети VLAN, добавление интерфейса VLAN в каждый шлюз, затем подключение виртуальных машин к этому шлюзу.

Использование шлюза представляет собой один (весьма часто применяемый на практике) способ создания сетевой среды для виртуальных машин, но это далеко не единственный способ. Ниже в текущем приложении в разделе «Использование Open vSwitch» будет рассматриваться применение Open vSwitch (OVS) для создания сетевой среды для виртуальных машин. Но сначала необходимо рассмотреть другой способ формирования сетевых соединений для виртуальных машин: интерфейсы macvtap.

Использование интерфейсов macvtap

В разделе «Использование интерфейсов macvlan» было показано, как применять интерфейсы macvlan для конфигурирования системы Linux с несколькими сетевыми объектами на одном физическом интерфейсе. Интерфейсы macvtap в определенной степени связаны с интерфейсами macvlan (они используют один и тот же драйвер ядра Linux), что позволяет использовать их при создании сетевой среды для виртуальных машин.

Чтобы применять интерфейсы macvtap вместе с KVM и Libvirt, необходимо снова начать с определения виртуальной сети Libvirt, которая содержит интерфейсы macvtap. Следующий фрагмент кода на языке XML определяет виртуальную сеть macvtap-net, поддерживающую интерфейсы macvtap, работающие в режиме шлюза и связанные с физическим интерфейсом eth1:

Tlgm: @it_boooks


```
<network>
  <name>macvtap-net</name>
  <forward mode="bridge">
    <interface dev="eth1"/>
  </forward>
</network>
```

Для шлюза с применением KVM и Libvirt конфигурация домена на языке XML должна содержать запись, указывающую на виртуальную сеть Libvirt:

```
<interface type="network">
  <source network="macvtap-net"/>
</interface>
```

После этого при запуске виртуальной машины с помощью Libvirt автоматически создается интерфейс macvtap на указанном (связанном) физическом интерфейсе. Это можно проверить, выполнив команду `ip link list`, в выводе которой вы должны увидеть интерфейс macvtap.

При использовании интерфейсов macvtap возникает один интересный побочный эффект: MAC-адрес, видимый в гостевом домене, полностью идентичен MAC-адресу, присвоенному интерфейсу macvtap. Ниже приведен пример фрагмента вывода команды `ip link list eth0` из гостевого домена при использовании интерфейса macvtap:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:9c:51:74 brd ff:ff:ff:ff:ff:ff
```

Для сравнения приведем вывод команды `ip link list macvtap0` на хост-системе, где `macvtap0` – интерфейс macvtap, созданный средствами Libvirt при активизации гостевого домена:

```
5: macvtap0@eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
    UNKNOWN mode DEFAULT group default qlen 500
    link/ether 52:54:00:9c:51:74 brd ff:ff:ff:ff:ff:ff
```

Очевидная прямая связь между MAC-адресом внутри гостевого домена и MAC-адресом за пределами этого домена может существенно упростить решение проблем и снизить трудозатраты при сборе текущей информации.

Далее будет рассматриваться еще один способ создания сетевой среды для виртуальных машин (с применением Open vSwitch), но прежде необходимо сделать краткий обзор использования нескольких важных сетевых функциональных возможностей ОС Linux.

Работа с сетевыми пространствами имен

Сетевые пространства имен (network namespaces) в ОС Linux представляют собой способ поддержки нескольких отдельных таблиц маршрутизации и нескольких отдельных конфигураций iptables, то есть способ ограничения «видимости» сетевых интерфейсов некоторым конкретным пространством имен. Возможно, это в наибольшей степени похоже на методику создания экземпля-

Tlgm: @it_boooks

ров виртуальной маршрутизации и переадресации (Virtual Routing and Forwarding – VRF) в сетевой среде, но сетевые пространства имен используются более разнообразными способами. Мы рассмотрим один из заслуживающих внимания подходов, в котором сетевые пространства имен применяются в совокупности с контейнерами Linux.



Несмотря на то что сетевые пространства имен можно использовать для создания экземпляров VRF, в сообществе разработчиков ядра Linux существует отдельная рабочая группа, которая в настоящее время занимается созданием «правильной» функциональности VRF в ядре Linux. Предполагается, что настоящая функциональность VRF обеспечит дополнительный логический уровень разделения (Layer 3) в пространстве имен. Пока еще рано делать выводы о том, к чему это приведет, но мы считаем, что наши читатели должны знать об этой разработке.

В каждой системе Linux при установке по умолчанию создается сетевое пространство имен, в котором пользователь может видеть таблицу маршрутизации, конфигурацию сетевого экрана iptables и сетевые интерфейсы. Но в этом разделе будет показано, что можно создавать и другие сетевые пространства имен и распределять сетевые интерфейсы по сетевым пространствам имен, отличающихся от заданных по умолчанию, для различных целей.

Начнем с изучения нескольких практических примеров применения сетевых пространств имен, которые помогут лучше понять возможности их использования.

Практические примеры использования сетевых пространств имен

Ниже перечислены некоторые возможные варианты применения сетевых пространств имен, которые могут оказаться полезными для вас:

- отдельная маршрутизация для каждого процесса – запуск процесса в собственном сетевом пространстве имен позволяет конфигурировать маршрутизацию для каждого процесса в отдельности;
- возможность создания конфигураций VRF – в начале раздела было отмечено, что сетевые пространства имен, вероятно, в наибольшей степени связаны с экземплярами VRF, поэтому вполне естественно, что одним из часто применяемых вариантов использования сетевых пространств имен является создание VRF-подобных конфигураций;
- поддержка пересекающихся пространств IP-адресов – возможно применение сетевых пространств имен для поддержки пересекающихся пространств IP-адресов, когда один и тот же адрес (или диапазон адресов) можно использовать для различных целей и для придания им различных значений. В более обобщенной схеме может потребоваться объединение этой методики с оверлейной сетевой структурой и/или с механизмом преобразования сетевых адресов (network address translation – NAT), чтобы обеспечить полную поддержку такого варианта.

Tlgm: @it_boooks

Продолжим рассмотрение практического применения сетевых пространств имен и начнем с того, как создаются (и удаляются) сетевые пространства имен, не определенные по умолчанию.

Создание и удаление сетевых пространств имен

Создание сетевых пространств имен выполняется чрезвычайно просто. Для этого снова воспользуемся утилитой `ip` из пакета `iproute2`, но сейчас будет применяться набор подкоманд `netns`.

Общий синтаксис команды создания сетевого пространства имен: `ip netns add` *`имя_пространства-имен`*. Ниже приведен пример создания сетевого пространства имен `blue`:

```
[vagrant@centos ~]$ ip netns add blue
[vagrant@centos ~]$
```

Здесь нет какого-либо сообщения об успешном выполнении команды, поэтому для проверки существования нового сетевого пространства имен выполняется следующая команда:

```
[vagrant@centos ~]$ ip netns list
blue
[vagrant@centos ~]$
```

Удаление сетевого пространства имен также не вызывает никаких затруднений:

```
[vagrant@centos ~]$ ip netns del blue
[vagrant@centos ~]$ ip netns list
[vagrant@centos ~]$
```

Отсутствие выводимых данных команды `ip netns list` свидетельствует об отсутствии сетевых пространств имен, кроме определенного по умолчанию пространства имен, в котором обычно находятся все сетевые объекты.

Теперь вам известно, как создаются и удаляются сетевые пространства имен, но гораздо важнее уметь ими пользоваться на практике. Сначала необходимо понять, как назначаются интерфейсы для конкретного пространства имен.

Размещение интерфейсов в сетевом пространстве имен

По умолчанию все объекты и конфигурации, так или иначе связанные с сетью, принадлежат сетевому пространству имен, определенному по умолчанию (оно обозначено как «netns 0»). Кроме того, по умолчанию новое создаваемое сетевое пространство имен не содержит сетевых интерфейсов. Таким образом, в только что созданном новом сетевом пространстве имен не установлено никаких соединений с какими-либо объектами: ни с сетевым пространством имен по умолчанию, ни с внешним миром, то есть полная изоляция. Чтобы исправить эту ситуацию, необходимо поместить интерфейс в новое сетевое пространство имен.

Tlgm: @it_boooks

Для размещения интерфейса в сетевом пространстве имен используется команда `ip link` (для выполнения примера предположим, что предварительно было создано новое сетевое пространство имен `blue`):

```
vagrant@jessie:~$ ip link set eth1 netns blue
vagrant@jessie:~$
```

Из этого примера видно, что общий синтаксис команды размещения интерфейса в сетевом пространстве имен выглядит следующим образом: `ip link set` *имя-интерфейса* `netns` *имя-пространства-имен*.

После помещения интерфейса в новое пространство имен он исчезает из пространства имен, определенного по умолчанию. Это разумно, так как в любой рассматриваемый момент времени интерфейс может существовать только в одном пространстве имен. Рассмотрим рис. А.1 и А.2. На рис. А.1 показан вывод команды `ip link list` в системе CentOS 7 с двумя физическими интерфейсами (ens32 и ens 33).

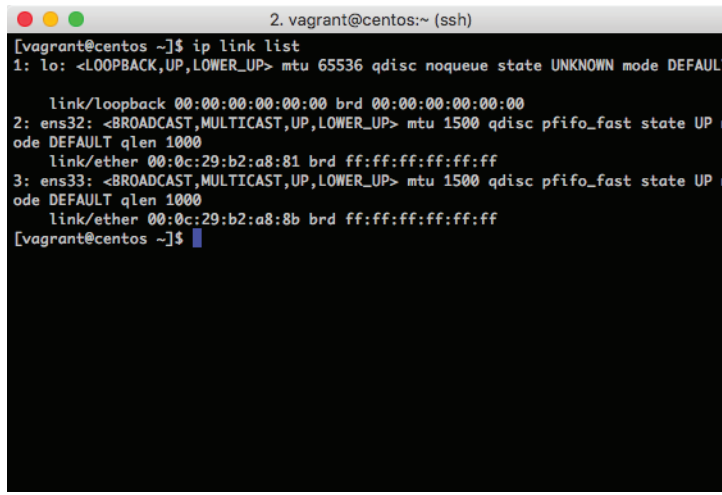


Рис. А.1 ❖ Вывод списка интерфейсов до размещения одного из интерфейсов в другом пространстве имен

Теперь рассмотрим рис. А.2, на котором показан вывод команды `ip link list` в той же системе CentOS 7, после того как один из физических интерфейсов был перемещен в другое сетевое пространство имен.

Здесь можно видеть, что интерфейс исчез из пространства имен, определенного по умолчанию.

Тут приведен только пример размещения физического интерфейса в альтернативном пространстве имен, но нет никаких ограничений для других типов интерфейсов. Предположим, что необходимо поместить интерфейс VLAN в новое пространство имен:

Tlgm: @it_boooks

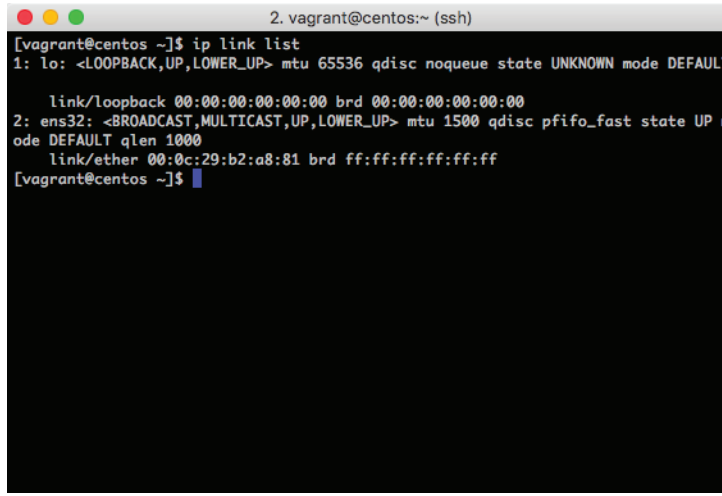


Рис. А.2 ❖ Вывод списка интерфейсов после перемещения одного из интерфейсов в другое пространство имен

```
[vagrant@centos ~]$ ip link set ens33.150 netns blue
[vagrant@centos ~]$
```

Допустим, что нужно поместить интерфейс macvlan в другое сетевое пространство имен:

```
[vagrant@centos ~]$ ip link set macvlan0 netns red
[vagrant@centos ~]$
```

Таким образом обеспечивается высокая степень гибкости при установлении соединений сетевых пространств имен с внешним миром.

Вне зависимости от типа интерфейса команда его размещения в некотором пространстве имен остается неизменной: `ip link set` *имя-интерфейса* `netns` *имя-пространства-имен*. И независимо от типа интерфейса после его назначения в альтернативное пространство имен этот интерфейс исчезает из пространства имен по умолчанию. Для работы с любым интерфейсом, помещенным в пространство имен, отличающееся от заданного по умолчанию, необходимо выполнять команды с явным указанием контекста пространства имен, которому принадлежит этот интерфейс. Иначе говоря, команды необходимо выполнять в контексте требуемого пространства имен.

Выполнение команд в определенном сетевом пространстве имен

Для выполнения любых команд в контексте определенного сетевого пространства имен используется специальная команда `ip netns exec`. Общий синтаксис: `ip netns exec` *пространство-имен команда*. Рассмотрим несколько примеров.

Tlgm: @it_boooks

В предыдущем разделе команда `ip link set` использовалась для передачи интерфейса eth1 в пространство имен blue в системе Debian 8.x. Если требуется проверить состояние этого интерфейса, то необходимо объединить команду `ip netns exec` (для выполнения требуемой команды в конкретном пространстве имен) с командой `ip link list` (для вывода списка сетевых интерфейсов) следующим образом:

```
vagrant@jessie:~$ ip netns exec blue ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
    default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

По выведенным данным можно определить, что интерфейс eth1 существует в пространстве имен `blue`, но в текущий момент отключен (запрещен), о чем свидетельствует элемент `state DOWN`. Для подключения (разрешения) этого интерфейса выполняется следующая команда:

```
vagrant@jessie:~$ ip netns exec blue ip link set eth1 up
vagrant@jessie:~$ ip netns exec blue ip link list eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
vagrant@jessie:~$
```

Теперь интерфейс активен и можно присвоить ему IP-адрес, затем проверить таблицу маршрутизации соответствующего пространства имен:

```
vagrant@jessie:~$ ip netns exec blue ip addr add 192.168.100.10/24 dev eth1
vagrant@jessie:~$ ip netns exec blue ip route list
192.168.100.0/24 dev eth1  proto kernel  scope link  src 192.168.100.11
vagrant@jessie:~$
```

Для подтверждения того, что пространства имен действительно отделены друг от друга, иначе говоря, что IP-конфигурация в пространстве имен `blue` не оказывает никакого воздействия на пространство имен по умолчанию, выполняется команда `ip route list` в пространстве имен по умолчанию, как показано ниже:

```
vagrant@jessie:~$ ip route list
default via 192.168.70.2 dev eth0
192.168.70.0/24 dev eth0  proto kernel  scope link  src 192.168.70.242
vagrant@jessie:~$
```

IP-конфигурация и назначенный маршрут, связанный с интерфейсом eth1, теперь никоим образом не влияет на пространство имен по умолчанию, так как функционирует в пространстве имен `blue`, в которое интерфейс eth1 был переведен. (В качестве самостоятельного упражнения предлагаем читателям проверить таблицу маршрутизации пространства имен `blue`.)

Tlgm: @it_boooks

После того как интерфейс переведен в альтернативное пространство имен, активизирован и для него сконфигурирован IP-адрес, можно протестировать соединение из пространства имен `blue` с внешней средой с помощью все той же команды `ip netns exec` и незаменимой команды `ping`:

```
vagrant@jessie:~$ ip netns exec blue ping -c 4 192.168.100.100
```

Во всех приведенных выше примерах вы могли заметить, что для выполнения в конкретном пространстве имен перед действительно выполняемыми командами всегда обязательно вводится часть `ip netns exec`. Можно воспользоваться функциональными свойствами алиасов (alias) командной оболочки bash, то есть возможностью создавать псевдокоманды, которые ссылаются на другие команды. Это очень удобная и полезная возможность. Например, можно определить алиас `nsblue` для выполнения команд в сетевом пространстве имен `blue`:

```
vagrant@trusty:~$ alias nsblue="ip netns exec blue"
vagrant@trusty:~$
```

После этого появляется возможность вводить более короткий алиас `nsblue` вместо `ip netns exec blue`, когда необходимо выполнять команды в сетевом пространстве имен `blue`.

```
vagrant@trusty:~$ nsblue ip link list
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
vagrant@trusty:~$
```

В приведенных выше примерах показаны передача только физического интерфейса в альтернативное сетевое пространство имен и работа с ним, но следует помнить, что можно передавать в альтернативное сетевое пространство имен интерфейсы практически любых типов – физические интерфейсы, интерфейсы VLAN, интерфейсы macvlan и т. д. При передаче интерфейса любого типа в альтернативное сетевое пространство имен вы устанавливаете соединение этого пространства имен с внешней средой (например, с конкретной виртуальной сетью VLAN, если перемещается интерфейс VLAN). А если необходимо установить соединение между новым пространством имен и пространством имен по умолчанию? В этом случае на помощь приходит методика использования пар veth (виртуальный Ethernet).

Соединение сетевых пространств имен с помощью пар veth

Пара виртуальный Ethernet (Virtual Ethernet; чаще обозначается термином «пара veth») представляет собой особый тип логического интерфейса, поддерживаемого ядром Linux. Название обусловлено тем, что виртуальные интерфейсы всегда работают в паре: трафик с одного интерфейса пары направляется на второй интерфейс той же пары. Как и другие типы интерфейсов, один ин-

Tlgm: @it_boooks

терфейс из пары veth может быть размещен в сетевом пространстве имен, отличающемся от заданного по умолчанию. Это позволяет пользователям устанавливать соединение между различными сетевыми пространствами имен.

Рассмотрим, как это работает. Сначала создается пара veth с помощью команды `ip`, синтаксис которой: `ip link add` *`имя-veth`* `type veth peer name` *`пара-veth`*. Если нужно создать пару veth с именами интерфейсов `veth0` и `veth1`, то команда должна выглядеть следующим образом:

```
vagrant@trusty:~$ ip link add veth0 type veth peer name veth1
vagrant@trusty:~$ ip -d link list veth0
5: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
    default qlen 1000
    link/ether f6:67:c0:f8:75:7d brd ff:ff:ff:ff:ff:ff promiscuity 0
    veth
vagrant@trusty:~$
```

Любой трафик, входящий на один из интерфейсов пары veth, будет направляться на второй интерфейс этой пары. Если поместить интерфейс veth1 в альтернативное сетевое пространство имен, то трафик, входящий на veth1 в любом пространстве имен, будет направлен на интерфейс veth0, расположенный в пространстве имен по умолчанию. Таким образом, устанавливается соединение между двумя сетевыми пространствами имен.

В приведенной ниже последовательности команд создается сетевое пространство имен `green`, затем интерфейс veth1 помещается в это пространство имен. После этого выполняется конфигурирование интерфейса veth1 с помощью команды `ip netns exec`. Далее производится тестирование соединения между двумя сетевыми пространствами имен.

```
vagrant@trusty:~$ ip netns add green
vagrant@trusty:~$ ip link set veth1 netns green
vagrant@trusty:~$ ip netns exec green ip addr add 10.0.3.1/24 dev veth1
vagrant@trusty:~$ ip netns exec green ip link set veth1 up
vagrant@trusty:~$ ip addr add 10.0.3.2/24 dev veth0
vagrant@trusty:~$ ip link set veth0 up
vagrant@trusty:~$ ping -c 4 10.0.3.1
PING 10.0.3.1 (10.0.3.1) 56(84) bytes of data.
64 bytes from 10.0.3.1: icmp_seq=1 ttl=64 time=0.046 ms
64 bytes from 10.0.3.1: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 10.0.3.1: icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from 10.0.3.1: icmp_seq=4 ttl=64 time=0.077 ms

--- 10.0.3.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.046/0.066/0.078/0.016 ms
vagrant@trusty:~$
```

Что здесь происходит? Пара veth уже была создана ранее, поэтому необходимо создать сетевое пространство имен `green` и поместить в него интерфейс veth1. Затем интерфейсу veth1 присваивается IP-адрес, и этот интерфейс активизируется (подключается). Чтобы в пространстве имен по умолчанию со-

Tlgm: @it_boooks

держался маршрут к цели, добавляется IP-адрес для интерфейса veth0. После этого с помощью утилиты `ping` выполняется тестирование соединения между двумя сетевыми пространствами имен.

А если необходимо установить соединение сетевого пространства имен с внешней средой с применением пары veth? Это не вызывает никаких затруднений – создается пара veth, один из интерфейсов этой пары размещается в требуемом пространстве имен, второй интерфейс пары veth помещается в шлюз с физическим интерфейсом. В результате сетевое пространство имен связано через шлюз с внешней средой. (Установление такого соединения предлагается выполнить в качестве самостоятельного учебного примера.)

Разумеется, можно создавать более сложные топологические схемы, но здесь мы описали лишь основные возможности использования пар veth для установления соединения между сетевыми пространствами имен.

В следующем разделе рассматривается еще один вариант практического применения сетевых пространств имен: контейнеры Linux.

Использование контейнеров Linux в сетевой среде

В предыдущем разделе рассматривались сетевые пространства имен как способ ограничения «видимости» сетевых интерфейсов только для некоторого подмножества компонентов системы, то есть изоляция сетевых интерфейсов в собственной небольшой окружающей среде (пространстве имен). Ядро Linux поддерживает и другие типы пространств имен: пространство имен монтируемых устройств, пространство имен процесса, а также (в самых новых версиях ядра) пользовательское пространство имен. В каждом случае основной целью пространства имен является ограничение видимости или ограничение ресурсов в изолированной собственной среде «суб-ОС». При объединении функциональности пространств имен с другими функциями уровня ядра Linux, такими как группы управления (control groups – cgroups), предоставляется возможность создания упрощенных «песочниц» (sandboxes), в которых процессы (или группы процессов) изолируются друг от друга. Такой подход является основой для контейнеров (containers) Linux, представляющих собой удобный и простой способ организации выполнения нескольких полностью изолированных друг от друга процессов в одной системе Linux.

Механизм контейнеров Linux существует уже достаточно давно, но только в последние несколько лет контейнеры стали пользоваться повышенным вниманием. В определенной степени этому способствовало появление проекта с открытым исходным кодом Docker – одна из моделей реализации работы с контейнерами Linux. В Docker используются пространства имен, cgroups и многоуровневая файловая система типа copy-on-write (копирование при записи) в совокупности с простым в использовании инструментом командной строки, что позволяет с чрезвычайной легкостью создавать и использовать контейнеры Linux.

Tlgm: @it_boooks

Если вас интересует более подробная информация о проекте Docker, то мы рекомендуем ознакомиться с книгой Эдриена Моуэта (Adrian Mouat) «Использование Docker»[1].

Но Docker – это не только наилучший вариант использования контейнеров. Ранее предлагался подход (некоторые считают его более продуманным) под названием LXC (аббревиатура, образованная от фразы LinuX Containers). Docker и LXC используют абсолютно одинаковые функциональные возможности ядра (пространства имен для изоляции и cgroups для распределения и ограничения ресурсов), но отличаются способами создания самих контейнеров и механизмами, предлагаемыми пользователям для работы с контейнерами. Каждый подход имеет свои преимущества и недостатки.

Помимо использования одних и тех же конструкций и свойств ядра Linux, проекты LXC и Docker объединяют определенные аспекты поддержки работы контейнеров в сетевой среде (все свойства и характеристики устанавливаются по умолчанию):

- LXC и Docker используют пары veth, размещая один из интерфейсов veth в альтернативном сетевом пространстве имен, в то время как второй интерфейс veth остается в сетевом пространстве имен по умолчанию;
- LXC и Docker используют шлюз Linux, к которому подключен интерфейс пара veth, размещенный в пространстве имен по умолчанию. В LXC шлюз по умолчанию получает имя lxcbr0, в Docker шлюз по умолчанию именуется docker0;
- LXC и Docker используют собственные специализированные правила сетевого экрана iptables для преобразования сетевых адресов (NAT) с целью обеспечения установления сетевых соединений для контейнеров.

Все перечисленные механизмы, за исключением правил сетевого экрана iptables, подробно рассматривались в предыдущих разделах, поэтому вам уже хорошо известны интерфейсы veth, способ размещения интерфейсов veth в различных сетевых пространствах имен и применение шлюза для обеспечения установления сетевых соединений.

Несмотря на то что LXC и Docker обладают многими весьма схожими свойствами, между ними имеются и некоторые различия, особенно в способах конфигурирования параметров и характеристик сетевой среды для контейнеров. Рассмотрим более подробно особенности конфигурирования сетевой среды и для LXC, и для Docker.

Конфигурирование сетевой среды в LXC

В LXC хранение сетевой конфигурации основано на принципе «отдельная конфигурация для каждого контейнера». В системах Ubuntu (путевое имя может отличаться в других дистрибутивах) файл */var/lib/lxc/<container-name>/config* содержит некоторые самые важные параметры и характеристики сетевой конфигурации для контейнера LXC:

---

[1] *Моуэт Э.* Использование Docker. М.: ДМК Пресс, 2017. ISBN 978-5-97060-426-7. 354 с.

Tlgm: @it_boooks

- параметр `lxc.network.type` управляет типом сетевой среды для контейнеров. По умолчанию этому параметру присваивается значение `veth`, то есть определяется использование пары veth. Также можно присвоить значение `macvlan`, чтобы использовать в контейнере LXC интерфейсы macvlan. В этом случае параметр `lxc.network.macvlan.mode` позволяет установить режим (`private`, `vepa`, `bridge`) для используемого интерфейса macvlan. Кроме того, поддерживается значение `vlan`, при установке которого контейнеры будут использовать интерфейс VLAN для установления соединений;
- параметр `lxc.network.link` управляет шлюзом, с которым LXC устанавливает соединение в сетевом пространстве имен по умолчанию. По умолчанию этому параметру присваивается значение `lxcbr0`. Если присваивается пустое значение, то контейнер не будет устанавливать соединение со шлюзом. В разделе «Использование Open vSwitch» будет показан пример с использованием пустого значения для этого параметра;
- параметр `lxc.network.veth.pair` определяет имя пары veth, которое будет известно за пределами пространства имен контейнера (один элемент пары будет помещен в сетевое пространство имен контейнера). Это позволит управлять соглашением об именах, используемых для пары veth, которая остается в сетевом пространстве имен по умолчанию;
- параметры `lxc.network.ipv4`, `lxc.network.ipv4.gateway`, `lxc.network.ipv6` и `lxc.network.ipv6.gateway` управляют конфигурацией IPv4 и IPv6 соответственно.

Даже по краткому описанию этих нескольких параметров можно понять, что LXC предоставляет достаточно обширный диапазон средств управления сетевой средой, формируемой для контейнеров.

Конфигурирование сетевой среды в Docker

Конфигурирование сетевой среды в Docker одновременно можно назвать и более простым, и более сложным, чем конфигурирование сетевой среды в LXC. Это может показаться странным и требует более подробного объяснения, которое приведено ниже.

До выхода версии 1.9 Docker использовались исключительно интерфейсы veth, шлюз Linux и набор правил iptables. Конфигурирование любого из этих компонентов подразумевало изменение значения переменной среды `DOCKER_OPTS`, которая использовалась при запуске демона Docker. Это простая часть процесса конфигурации.

В версии 1.9 в Docker была добавлена динамически подключаемая сетевая подсистема, которая позволяла создавать несколько различных типов сетей, управляемых демоном Docker. «Старая» сеть Docker с применением шлюза осталась доступной, но появилась возможность создавать оверлейные мультихостовые сети. Кроме того, появилась поддержка подключаемых сетевых модулей от сторонних производителей.

Tlgm: @it_boooks

Следующее изменение сетевой среды Docker произошло в версии 1.12, где появился новый режим Swarm mode и встроенная оверлейная мультихостовая виртуальная сеть VXLAN. При этом потребовалось заново переписать подключаемые сетевые модули от сторонних производителей, и этот процесс пока еще не завершен (на момент написания данной книги).



Поддерживаются ли интерфейсы ipvlan?

В этом разделе рассматривались различные типы логических интерфейсов: например, интерфейсы VLAN, интерфейсы macvlan, пары veth. Не упоминались интерфейсы ipvlan, которые похожи на интерфейсы macvlan, но отличаются тем, что используют IP-адреса на уровне 3, а не MAC-адреса на уровне 2. Поддержка интерфейсов ipvlan пока еще остается нововведением, тем не менее уже известны реализации вариантов использования интерфейсов ipvlan в сетевой среде контейнеров.

Поскольку функциональность многохостовой сетевой среды Docker продолжает изменяться очень быстро, мы сосредоточимся на определяемой по умолчанию сети с использованием шлюза для контейнеров Docker. Почти все конфигурации принятой по умолчанию в Docker сети с использованием шлюза подразумевают изменение значений флагов, передаваемых в демон Docker при его запуске. Это осуществляется посредством внесения изменений в значение переменной среды `DOCKER_OPTS` или посредством редактирования файла, используемого при запуске демона. Некоторые наиболее употребляемые параметры и ключи конфигурации описаны ниже:

- параметр `-b` или `--bridge` позволяет определить имя шлюза, который будет использовать Docker. Для этого параметра можно задать имя, отличающееся от `docker0`, принятого по умолчанию;
- параметр `--bip` позволяет определить IP-адрес, присваиваемый интерфейсу шлюза;
- параметр `--iptables=true` позволяет в сетевой среде Docker автоматически добавлять соответствующие правила iptables для выполнения NAT.

В дополнение к этим параметрам, предназначенным для демона (то есть для всех созданных контейнеров), существуют параметры, специализированные для отдельных контейнеров, в частности это определения портов, открытых во внешнюю окружающую среду. Определения открытых для внешнего доступа портов можно найти в следующих двух вариантах:

- файл `Dockerfile` для конкретного контейнера может содержать определение одного или нескольких портов в команде `EXPOSE`, которая сообщает демону Docker, какие именно порты необходимо открыть для доступа из внешней среды. Например, команда `EXPOSE 80` в `Dockerfile` оповещает демон Docker о необходимости установления динамической связи общедоступного порта на хосте Docker с портом 80 в контейнере. Здесь очень важно отметить, что назначаемый общедоступный порт хоста не является строго определенным и выбирается динамически демоном Docker;

Tlgm: @it_boooks

- команда запуска контейнера может содержать один или несколько параметров `-p`, которые позволяют пользователю управлять установлением связей между общедоступными портами хоста и портами контейнера. Например, параметр `-p 80:80` сообщает демону Docker о необходимости установления соединения между портом 80 на хосте Docker и портом 80 в запускаемом контейнере.

К счастью, Docker предлагает команду `docker inspect`, которая (если известен идентификатор ID активного работающего контейнера) предоставляет всю необходимую информацию о сетевой конфигурации контейнера, включая IP-адрес контейнера и данные об установленной связи портов.

После рассмотрения сетевой среды для контейнеров Docker и LXC перейдем к последней важной теме: использование Open vSwitch в сетевой среде.

Использование Open vSwitch

Open vSwitch (OVS) – проект с открытым исходным кодом, предлагающий высококачественную реализацию многоуровневого виртуального коммутатора для работы в гипервизоре (хотя, как мы убедимся несколько позже, для OVS возможны многочисленные варианты использования не только под управлением гипервизора). OVS изначально был предназначен для автоматизации сети и обеспечивал программное управление, сохраняя при этом поддержку широкого диапазона управляющих протоколов и стандартов. Кроме того, в OVS с самого начала предусматривалась поддержка OpenFlow и основного протокола SDN, поэтому OVS рассматривался многими специалистами как наиболее полная реализация OpenFlow. OVS имеет широкую поддержку: гипервизоры Xen и KVM поддерживают OVS, а на момент написания данной книги было почти завершено портирование OVS в среду Hyper-V (вероятно, этот процесс будет завершен полностью к моменту издания книги). Многочисленные системы управления и оркестровки, в том числе OpenStack, также поддерживают OVS.

С учетом весьма заметной роли в областях применения SDN и автоматизации сети вполне можно ожидать достаточно полного описания OVS в книге об автоматизации сети. Но OVS предоставляет слишком большой объем функциональных возможностей и характеристик, поэтому объем рассматриваемого здесь материала ограничен. Мы сосредоточимся на трех главных темах:

- установка OVS (только в ОС Linux);
- конфигурирование OVS;
- соединение нескольких типов рабочих нагрузок в OVS.

Начнем, разумеется, с установки OVS.

Установка OVS

Архитектура OVS, объединяющая демона в пользовательском пространстве и модуль ядра, определяет различные процедуры установки OVS в зависимо-

Tlgm: @it_boooks

сти от конкретного дистрибутива Linux и версии модуля ядра OVS, который будет использоваться.

В версию 3.3 и в более поздние версии ядра Linux включен модуль OVS. Для его использования не требуется каких-либо дополнительных действий, необходимо установить только компоненты для пользовательского пространства. Но если вы предпочитаете пользоваться модулем ядра непосредственно из дерева версий OVS (возможно, более новым, чем штатный модуль, следовательно, поддерживающим дополнительные функциональные возможности), то потребуются загрузка и компиляция такого модуля для текущей используемой версии ядра Linux.



Если нужно проверить, поддерживает ли ваше ядро Linux модуль OVS, то просто выполните команду `modprobe openvswitch` (возможно, потребуется префикс sudo для временного получения прав суперпользователя). Если команда сообщает об ошибке, то в ядре отсутствует модуль OVS, следовательно, необходимо его установить. Но помните, что штатный модуль OVS включен в ядро с версии 3.3, поэтому практически во всех современных дистрибутивах модуль OVS должен быть доступен сразу после установки системы.

В системах Debian 8.x и Ubuntu 14.04 пакеты компонентов для пользовательского пространства и модуля ядра доступны в основных репозиториях. Таким образом, установка сводится к выполнению команды `apt-get install`:

- для использования штатного модуля ядра установите только пакеты для пользовательского пространства с именами `openvswitch-common` и `openvswitch-switch`;
- для использования модуля ядра из комплекта OVS необходимо также установить пакет `openvswitch-datapath-dkms` (и требуемые для него пакеты `dkms`, `make` и `libc6-dev`).



Выше было отмечено, что пакеты OVS для Debian 8.x находятся в основных репозиториях, но следует всегда помнить о том, что в зависимости от применяемого метода установки основные репозитории не всегда могут быть включены в список разрешенных репозиториев. Проверьте конфигурацию репозиториев в файле /etc/apt/sources.list, если не уверены в том, что все необходимые репозитории разрешены (можно воспользоваться командой `cat` для просмотра конфигурации, потом отредактировать список разрешенных репозиториев при необходимости).

В системах RHEL/CentOS/Fedora процедура установки несколько сложнее. В разрешенных репозиториях RHEL 7.x и CentOS 7.x не содержатся пакеты OVS. Чтобы установить OVS, потребуется скомпилировать его из исходных кодов, или добавить репозиторий, в котором содержатся пакеты OVS. Один из таких репозиториев – OpenStack в группе CentOS Cloud Special Interest Group (SIG). Включить этот репозиторий в список разрешенных можно командой `yum install centos-release-openstack`. После выполнения команды проверьте наличие нового репозитория в списке с помощью команды `yum repolist`. В системе Fedora репозиторий с пакетами OVS по умолчанию включен в список

Tlgm: @it_boooks

доступных при установке системы, поэтому не нужно добавлять дополнительные репозитории.

Для установки OVS в системах RHEL/CentOS/Fedora при доступности требуемых пакетов достаточно выполнить команду `yum install`, чтобы установить пакет `openvswitch`.

Отметим, что на момент написания данной книги в системах RHEL/CentOS/Fedora не предлагался пакет для установки модуля ядра из дерева версий OVS. Поэтому при необходимости использования такого модуля ядра его нужно вручную скомпилировать и установить. Ручная компиляция и установка модуля ядра – это отдельная обширная тема, которая не связана с темами, обсуждаемыми в данной книге. Но в различных общедоступных источниках можно найти ряд рекомендаций и правил по ручной компиляции и установке модулей ядра.

После того как все компоненты OVS установлены, можно переходить к конфигурированию OVS.

Конфигурирование OVS

OVS можно конфигурировать несколькими различными способами: использовать специально предназначенные для этого утилиты командной строки или редактировать скрипты, включенные в комплект сетевой конфигурации ОС Linux (*/etc/network/interfaces* в системах Debian/Ubuntu, */etc/sysconfig/network-scripts* в системах RHEL/CentOS/Fedora). В этом разделе основное внимание будет уделено использованию утилит командной строки, предназначенных для работы с OVS. Причина такого выбора заключается в том, что OVS не соблюдает общепринятое соглашение Linux о необходимости редактирования файлов конфигурации, для того чтобы конфигурация стала постоянной.

Изменения, вносимые в конфигурацию с помощью утилит командной строки OVS, становятся действительно постоянными. OVS поддерживает собственную базу данных конфигурации, а утилиты командной строки работают с этой базой данных. При перезагрузке системы OVS повторно считывает конфигурацию из своей базы данных, следовательно, любое изменение, внесенное с помощью утилит командной строки, действительно становится постоянным. Это главное отличие OVS от множества других сетевых конфигураций, которые рассматривались в главе 3 и в текущем приложении.

Основная утилита, используемая для конфигурирования OVS, – `ovs-vsctl`. Подобно команде `ip`, которой мы пользовались в этом приложении и в главе 3, команда `ovs-vsctl` предлагает набор подкоманд для различных целей:

- подкоманда `show` просто выводит обзор содержимого базы данных конфигурации (то есть обзор конфигурации OVS);
- подкоманда `add-br` добавляет шлюз в конфигурацию OVS. Любой OVS-шлюз концептуально и функционально похож на шлюз Linux, но обладает значительно расширенным набором функциональных возможностей;
- подкоманда `del-br` удаляет OVS-шлюз;

Tlgm: @it_boooks

- подкоманда `add-port` добавляет порт в OVS-шлюз. Порты могут быть физическими интерфейсами (например, eth1 или ens33) или логическими сетевыми интерфейсами (например, интерфейсом VLAN или интерфейсом veth);
- подкоманда `del-port` удаляет порт из OVS-шлюза.

В действительности подкоманд намного больше, но функциональности перечисленных выше подкоманд вполне достаточно, чтобы начать работу с OVS. Рассмотрим несколько примеров.

Предположим, что OVS уже установлен и запущен. Начнем с создания OVS-шлюза. Сначала выведем для просмотра текущую конфигурацию (она фактически пустая, поскольку OVS пока еще не сконфигурирован), затем добавим шлюз и снова выведем текущую конфигурацию. Синтаксис команды добавления шлюза в OVS: `ovs-vsctl add-br` *`имя-шлюза`*. Выполняемые команды выглядят следующим образом:

```
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    ovs_version: "2.4.0"
[vagrant@centos ~]$ ovs-vsctl add-br br0
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.4.0"
[vagrant@centos ~]$
```

Теперь OVS-шлюз существует, но, как и шлюз Linux, он не может выполнять какие-либо операции до тех пор, пока в него не будут добавлены порты. Добавим в этот шлюз физический интерфейс ens33:

```
[vagrant@centos ~]$ ovs-vsctl add-port br0 ens33
```

Из этого примера понятно, что синтаксис команды добавления порта в шлюз: `ovs-vsctl add-port` *`имя-шлюза имя-порта`*. Порт, добавляемый в OVS, обязательно должен существовать и распознаваться системой Linux, кроме единственного исключительного случая, который мы рассмотрим позже.

Теперь можно проверить, действительно ли физический порт был добавлен в OVS-шлюз:

```
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "ens33"
            Interface "ens33"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.4.0"
```

Tlgm: @it_boooks

Для удаления порта или шлюза используются подкоманды `del-port` или `del-br` соответственно:

```
[vagrant@centos ~]$ ovs-vsctl del-port br0 ens33
[vagrant@centos ~]$ ovs-vsctl del-br br0
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    ovs_version: "2.4.0"
[vagrant@centos ~]$
```

В дополнение к рассмотренным выше подкомандам может потребоваться подкоманда `set` для установки характеристик или значений параметров конфигурации. Например, для присваивания тега VLAN OVS-порту используется команда со следующим синтаксисом: `ovs-vsctl set port` *`имя-порта тег=значение`*. Предположим, что имеется порт vnet0, представляющий виртуальную машину (этот сценарий более подробно будет рассмотрен в следующем разделе «Использование виртуальных машин совместно с OVS»), и требуется, чтобы виртуальная машина стала виртуальной сетью VLAN 10. Для этого выполняется следующая команда:

```
vagrant@trusty:~$ ovs-vsctl set port vnet0 tag=10
vagrant@trusty:~$ ovs-vsctl show
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "vnet0"
            tag: 10
            Interface "vnet0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "eth1"
            Interface "eth1"
    ovs_version: "2.0.2"
vagrant@trusty:~$
```

Кроме того, полезной может оказаться подкоманда `list`, которая выводит список свойств и значений параметров, связанных с определенным объектом конфигурации, содержащимся в базе данных конфигурации OVS. Если необходимо просмотреть все свойства и значения параметров для порта vnet0, то выполняется следующая команда:

```
vagrant@trusty:~$ ovs-vsctl list port vnet0
_uuid               : cc51fc7e-ce14-41c6-9ad6-7b3ae717afa9
bond_downdelay      : 0
bond_fake_iface     : false
bond_mode           : []
bond_updelay        : 0
external_ids        : {}
fake_bridge         : false
interfaces          : [74e6ede7-1a13-45c1-84d6-f66cbfc5a353]
lacp                : []
```

Tlgm: @it_boooks


```
mac                 : []
name                : "vnet0"
other_config        : {}
qos                 : []
statistics          : {}
status              : {}
tag                 : 10
trunks              : []
vlan_mode           : []
vagrant@trusty:~$
```

Разумеется, возможности команды `ovs-vsctl` гораздо более широкие – создание оверлейных сетей с применением протоколов типа VXLAN или Geneve, работа с потоками OpenFlow, настройка OVS для использования внешнего контроллера и т. д., – но в основном для повседневной работы с OVS потребуются операции добавления и удаления шлюзов и портов, а также установка свойств и значений параметров для портов.

В следующем разделе будет рассматриваться соединение нескольких типов рабочих нагрузок в OVS.

Соединение нескольких типов рабочих нагрузок в OVS

Здесь мы будем использовать термин «рабочая нагрузка» (workload) для обозначения любого типа объекта, для которого требуется сетевое соединение. Это может быть сетевое пространство имен, контейнер, виртуальная машина (например, госевой домен KVM) или сама система хоста OVS.

Процесс соединения рабочих нагрузок в OVS основан на нескольких разнообразных предпосылках, но в основном можно выделить следущие:

- для сетевых пространств имен и контейнеров чаще всего используются пары veth при соединении сетевого пространства имен с OVS;
- для гостевых доменов KVM соединение с OVS обычно реализуется через интерфейс TAP;
- для системы хоста можно напрямую направлять трафик через OVS, используя для этого внутренний порт OVS.

Рассмотрим каждый из этих сценариев немного подробнее. Описания функциональности команд, используемых в следующих примерах, можно найти в предыдущих разделах.

Соединение сетевых пространств имен с OVS

В разделе о сетевых пространствах имен описывался способ соединения сетевых пространств имен с использованием пары veth. Один из интерфейсов veth размещается в сетевом пространстве имен (с помощью команды `ip link set`), другой интерфейс этой пары остается в основном пространстве имен.

Пары veth можно использовать для соединения сетевых пространств имен с OVS (следовательно, и с любым типом топологии сети, которую поддерживает OVS, – физическая сеть или оверлейная сеть). Для этого необходимо воспользоваться основной схемой настройки, которая была описана в пре-

Tlgm: @it_boooks

дыдущих разделах, чтобы создать шлюз из некоторого сетевого пространства имен в сеть.

Предположим, что существует сетевое пространство имен `green`. Сначала нужно создать пару veth, поместить один интерфейс veth в пространство имен `green` и сконфигурировать этот интерфейс в пространстве имен `green`:

```
[vagrant@centos ~]$ ip link add veth0 type veth peer name veth1
[vagrant@centos ~]$ ip link set veth1 netns green
[vagrant@centos ~]$ ip netns exec green ip addr add 192.168.100.12/24 dev veth1
[vagrant@centos ~]$ ip netns exec green ip link set veth1 up
[vagrant@centos ~]$ ip link set veth0 up
```

После этого у нас имеется пара veth (veth0 и veth1), при этом интерфейс veth1 перемещен в пространство имен green и ему присвоен IP-адрес. Оба интерфейса veth активизированы (разрешены), поэтому между ними возможна передача трафика. Для соединения сетевого пространства имен `green` с OVS необходимо просто добавить интерфейс veth0 в OVS-шлюз. Предположим, что OVS-шлюз br0 уже существует и содержит физический интерфейс ens33:

```
[vagrant@centos ~]$ ovs-vsctl add-port br0 veth0
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "veth0"
            Interface "veth0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "ens33"
            Interface "ens33"
    ovs_version: "2.4.0"
[vagrant@centos ~]$
```

Здесь можно видеть, что использована команда `ovs-vsctl add-port`, в которой задано имя шлюза `br0` и имя добавляемого интерфейса `veth0`. Теперь сетевое пространство имен `green` соединено с OVS (в этой конкретной конфигурации установлено соединение сетевого пространства имен через шлюз с сетью, соединенной с физическим интерфейсом ens33).

После установления соединения сетевого пространства имен с OVS появляется возможность использования всех функциональных преимуществ OVS. Но здесь мы ограничимся лишь простым примером, показанным выше.

Теперь рассмотрим пример, более приближенный к практике: использование контейнеров совместно с OVS.

Использование контейнеров совместно с OVS

Так как контейнеры используют сетевые пространства имен, многое из того, что рассматривалось в предыдущем разделе, можно применить и в нашем случае. Основное различие заключается в особенностях организации рабочего потока контейнеров.

Tlgm: @it_boooks

К моменту написания данной книги контейнеры Docker не имели встроенных методов для установления соединения с OVS в сетевой среде. Несмотря на то что Docker использует пары veth и может воспользоваться функциональностью шлюза Linux, а OVS поддерживает шлюзы, во многом похожие на шлюз Linux, прямое соединение контейнеров Docker с OVS пока еще не реализовано. Но в этой области все меняется очень быстро, и можно предположить с большой уверенностью, что прямая связь между контейнерами Docker и OVS будет обеспечена в ближайшем будущем.

С другой стороны, LXC обеспечивает поддержку OVS. Для реализации этой возможности существуют, по меньшей мере, два способа:

- если вместе с LXC используется среда (и библиотека) Libvirt, то можно воспользоваться виртуальной сетью Libvirt как внешним сетевым сегментом OVS-шлюза. Этот процесс будет описан в следующем разделе «Использование виртуальных машин совместно с OVS». Вне зависимости от того, используются контейнеры или виртуальные машины, работа с виртуальной сетью Libvirt будет выполняться одинаково;
- в другом варианте можно сконфигурировать LXC так, чтобы использовать скрипт для подключения одного из интерфейсов пары veth к OVS.

Рассмотрим более подробно второй вариант. (Здесь мы ограничимся только описанием конфигурирования LXC в системе Ubuntu.) Выше отмечалось, что по умолчанию LXC хранит информацию о конфигурации контейнеров в файлах */var/lib/lxc/<имя-контейнера>/config*, таким образом, именно в этом файле содержатся параметры конфигурации, необходимые для установления соединения LXC с OVS в сетевой среде. Основные параметры конфигурации были описаны выше, в разделе «Конфигурирование сетевой среды в LXC», но существует еще один параметр, который особенно важен в рассматриваемом здесь случае:

- параметр `lxc.network.script.up` определяет имя скрипта, который будет запускаться при активизации (подключении) сетевого интерфейса контейнера. Именно здесь можно определить скрипт, в котором создается пара veth (имя которой известно, так как оно управляется параметром `lxc.network.veth.pair`) и подключается к OVS-шлюзу. Простой пример такого скрипта может выглядеть приблизительно следующим образом:

```
#!/bin/bash

BRIDGE="br0"
ovs-vsctl --may-exist add-br $BRIDGE
ovs-vsctl --if-exists del-port $BRIDGE $5
ovs-vsctl --may-exist add-port $BRIDGE $5
```

Элемент `$5` ссылается на пятый параметр, передаваемый в скрипт, которым в данном случае является имя интерфейса veth, определяемое параметром конфигурации `lxc.network.veth.pair`. Мы не рассматривали ключи `--may-exist` или `--if-exists` для команды `ovs-vsctl`, но их смысл должен быть понятен из названий, а поведение вполне предсказуемо. Ключ `--may-exist` предотвращает

Tlgm: @it_boooks

ошибку, если шлюз или порт уже существует, а ключ `--if-exists` выполняет операцию только в том случае, если заданный объект существует.

При использовании этого варианта конфигурации LXC создает пару veth (с именами интерфейсов, определяемыми параметром конфигурации `lxc.network.veth.pair`), затем запускает предоставленный скрипт. Скрипт находит требуемый интерфейс veth и подключает его к заданному OVS-шлюзу. Таким образом, устанавливается соединение контейнера с OVS и с любыми сетевыми топологиями, которые поддерживаются конфигурацией OVS (например, шлюзовое или оверлейное соединение).

Но как использовать OVS совместно с виртуальными машинами? В следующем разделе будет показано, что совместное использование виртуальных машин и OVS также не вызывает никаких затруднений.

Использование виртуальных машин совместно с OVS

Для сохранения объема излагаемого материала в разумных пределах ограничимся (как и в предыдущих разделах) описанием совместной работы гипервизора KVM и OVS. Это не означает наличия каких-либо ограничений со стороны OVS, просто размер раздела (соответственно, и книги) нельзя увеличивать до бесконечности.

В разделе «Виртуальные машины в сетевой среде» в начале текущего приложения была представлена концепция виртуальной сети Libvirt, представляющая собой абстракцию Libvirt, используемую для обращения к структурам более низкого уровня. В последние несколько лет библиотека Libvirt обеспечила встроенную поддержку OVS, поэтому виртуальные сети Libvirt могут напрямую использовать OVS.

Ниже приведен небольшой фрагмент на языке XML, в котором определяется виртуальная сеть Libvirt с поддержкой OVS:

```
<network>
  <name>ovs-net</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
  <virtualport type="openvswitch"/>
</network>
```

В дальнейшем на эту виртуальную сеть можно ссылаться по имени в конфигурации гостевого домена KVM, как показано в следующем примере (здесь показана только часть конфигурации гостевого домена, относящаяся непосредственно к сетевой среде):

```
<interface type="network">
  <source network="ovs-net"/>
</interface>
```

При использовании этого типа конфигурации после начала работы гостевого домена KVM вы увидите новый интерфейс, соединенный с OVS, если выполните команду `ovs-vsctl show`:

Tlgm: @it_boooks


```
vagrant@trusty:~$ ovs-vsctl show
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "eth1"
            Interface "eth1"
        Port "br0"
            Interface "br0"
                type: internal
        Port "vnet0"
            Interface "vnet0"
    ovs_version: "2.0.2"
vagrant@trusty:~$
```

Это интерфейс TAP, который можно проверить командой `ip -d link list vnet0` (отметим в приведенном ниже выводе команды строку `tun`, которая означает, что это устройство TUN/TAP):

```
vagrant@trusty:~$ ip -d link list vnet0
7: vnet0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master
    ovs-system state UNKNOWN mode DEFAULT group default qlen 500
    link/ether fe:54:00:19:bc:6f brd ff:ff:ff:ff:ff:ff promiscuity 1
    tun
vagrant@trusty:~$
```

Теперь виртуальная машина соединена через шлюз с физической сетью, подключенной к интерфейсу eth1, но с помощью сетевых пространств имен вы можете воспользоваться любыми необходимыми функциональными возможностями OVS через установленное соединение.

До настоящего момента демонстрировались соединения сетевых пространств имен, контейнеров и виртуальных машин с OVS. А если необходимо направить через OVS трафик из системы хоста, на котором работает OVS? Для этого используется внутренний порт OVS.

Использование внутренних портов OVS

Внутренние порты OVS позволяют поместить представление логического сетевого интерфейса в стек TCP/IP хоста. С этой точки зрения можно сравнить внутренние порты OVS с интерфейсами VLAN, macvlan и veth – все они являются логическими сетевыми интерфейсами. Главное отличие состоит в том, что внутренние порты OVS существуют только в контексте конкретной конфигурации OVS.

Рассмотрим пример. В двух предыдущих разделах уже демонстрировалось использование OVS-шлюза br0. В каждом OVS-шлюзе всегда существует соответствующий внутренний порт. Вы уже видели его, хотя могли и не заметить. Рассмотрим следующий вывод команды `ovs-vsctl show`:

```
vagrant@trusty:~$ ovs-vsctl show
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "eth1"
```

Tlgm: @it_boooks


```
            Interface "eth1"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.0.2"
vagrant@trusty:~$
```

Отметим, что `br0` обозначен как порт и как интерфейс типа `internal`. Это и есть внутренний порт OVS, а факт вывода информации об этом интерфейсе командой `ip link list` подтверждает, что сетевой стек хоста распознает его как логический сетевой интерфейс.

```
vagrant@trusty:~$ ip link list br0
6: br0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN mode DEFAULT
    group default
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
vagrant@trusty:~$
```

Если удалить OVS-шлюз командой `ovs-vsctl del-br br0`, то какую информацию об этом интерфейсе выведет команда `ip link list`?

```
vagrant@trusty:~$ ip link list br0
Device "br0" does not exist.
vagrant@trusty:~$
```

Это подтверждает вышеприведенное утверждение о том, что внутренний порт OVS существует только в контексте конфигурации OVS. Он не является частью конфигурации сетевого стека хоста, только частью конфигурации OVS. При удалении внутреннего порта из OVS он автоматически удаляется из сетевой конфигурации хоста.

Этот факт можно использовать для воздействия на процесс управления трафиком в сетевом стеке хоста. Предположим, что необходимо создать логический сетевой интерфейс, который будет работать как конечный пункт тоннеля (tunnel endpoint – TEP) для оверлейного трафика VXLAN, управляемого OVS. Ниже приведены команды создания внутреннего порта OVS (они будут описаны после примера):

```
[vagrant@centos ~]$ ovs-vsctl add-port br0 tep0 -- set interface tep0 type=internal
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "ens33"
            Interface "ens33"
        Port "tep0"
            Interface "tep0"
                type: internal
    ovs_version: "2.4.0"
```

Tlgm: @it_boooks

Не совсем обычный синтаксис команд необходим, поскольку OVS предполагает, что при добавлении интерфейсов в OVS они уже существуют. В действительности интерфейс tep0 не существует, так как мы создаем его во время выполнения команды. Поэтому используется двойной дефис, чтобы сообщить OVS о необходимости объединения этих команд, то есть создается порт tep0 и одновременно устанавливается его тип `internal`.

Отметим, что эти команды можно выполнить по отдельности, если не обращать внимания на вывод сообщения об ошибке от OVS после первой команды:

```
[vagrant@centos ~]$ ovs-vsctl add-port br0 tep0
ovs-vsctl: Error detected while setting up 'tep0'. See ovs-vswitchd log for details.
[vagrant@centos ~]$ ovs-vsctl set interface tep0 type=internal
[vagrant@centos ~]$ ovs-vsctl show
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "ens33"
            Interface "ens33"
        Port "tep0"
            Interface "tep0"
                type: internal
    ovs_version: "2.4.0"
[vagrant@centos ~]$
```

Теперь интерфейс tep0 существует и можно заняться его конфигурацией точно так же, как для любого другого логического интерфейса. Ниже показаны присваивание IP-адреса интерфейсу tep0 и подключение (разрешение) этого интерфейса:

```
[vagrant@centos ~]$ ip link list tep0
10: tep0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether 9e:da:79:89:c3:6a brd ff:ff:ff:ff:ff:ff
[vagrant@centos ~]$ ip addr add 10.1.1.100/24 dev tep0
[vagrant@centos ~]$ ip link set tep0 up
[vagrant@centos ~]$ ip route list
default via 192.168.70.2 dev ens32  proto static  metric 100
10.1.1.0/24 dev tep0  proto kernel  scope link  src 10.1.1.100
192.168.70.0/24 dev ens32  proto kernel  scope link  src 192.168.70.244
192.168.70.0/24 dev ens32  proto kernel  scope link  src 192.168.70.244  metric
100 [vagrant@centos ~]$
```

В информации, выводимой командой `ip route list`, можно видеть, что на сетевую конфигурацию хоста оказывает воздействие конфигурация внутреннего порта OVS – в системе CentOS теперь существует новый маршрут, связанный с IP-адресом, который был присвоен интерфейсу tep0.

Теперь проверим, действительно ли вы понимаете, как работает эта конфигурация: как трафик с интерфейса tep0 попадает в сеть? Если вы сказали: через интерфейс ens33, то дали абсолютно правильный ответ. Внутренний ин-

Tlgm: @it_boooks

терфейс OVS – это логический интерфейс, связанный с физической сетью через шлюз br0, который содержит физический интерфейс ens33. Подобным же образом входящий трафик для подсети 10.1.1.100/24 будет передаваться в систему через интерфейс ens33.

В этом разделе рассматривался лишь начальный небольшой объем информации о функциональных возможностях OVS, но, по меньшей мере, вы получили общее представление об основных концепциях, заложенных в основу этого механизма. Как уже было отмечено ранее, OVS является ключевой частью многих широко известных проектов с открытым исходным кодом, поэтому время, затраченное на освоение работы с OVS, многократно окупится в будущем.

Tlgm: @it_boooks

Приложение Б

Использование NAPALM

NAPALM – Network Automation and Programmability Abstraction Layer with Multivendor support – это библиотека на языке Python, которая предлагает высококачественный профессиональный комплект операций для управления сетевыми устройствами с использованием общедоступного набора объектов языка Python независимо от того, как именно будет выполняться каждая операция для заданного конкретного типа устройства.

NAPALM обладает постоянно наращиваемым набором функциональных возможностей, но в этом разделе мы сосредоточим внимание только на двух основных его функциях:

- управление конфигурацией;
- получение информации от сетевых устройств.

Для каждой из этих функций отметим, что выполнение любой конкретной операции в одинаковой степени независимо от производителя оборудования или операционной системы, с которой вы работаете, если имеется поддерживаемый NAPALM драйвер и функциональная поддержка этой операции.

NAPALM поддерживает широкий спектр устройств от различных производителей и использует различные API для обмена информацией с каждым типом устройств. Например, Cisco Nexus в настоящее время использует NX-API, Arista EOS – eAPI, Cisco IOS – SSH, а драйверы Junuiper Junos используют NETCONF. Начиная работать с NAPALM, необходимо знать, какой API требуется для устройств, с которыми вы работаете.

За более подробными описаниями поддерживаемых API и устройств, а также по всем темам, которые не рассматриваются в этом приложении, обращайтесь к официальной документации NAPALM (https://napalm.readthedocs.io/en/latest/). Начнем с изучения управления конфигурацией с использованием NAPALM.

Управление конфигурацией с использованием NAPALM

NAPALM предлагает собственный, отличающийся от прочих подход к управлению конфиграциями сетевых устройств, но при этом сохраняет возможность использования более привычных общепринятых методик конфигурирования устройств. Единственный в своем роде подход NAPALM обычно обозначают

Tlgm: @it_boooks

термином «декларативное управление конфигурацией» (declarative configuration management).

По методике декларативной конфигурации все внимание сосредоточено на том, какой должна быть требуемая конкретная конфигурация устройства. Такой подход является полной противоположностью подходу, при котором сначала выясняется, что представляет из себя текущая конфигурация, затем начинается поиск путей перехода от текущего состояния к требуемому. Декларативная конфигурация – главное преимущество и основная функциональная особенность NAPALM, но в действительности этот подход поддерживается собственными функциональными возможностями сетевых устройств от различных производителей. Примерами таких зависимых от конкретного устройства функций являются конфигурации-кандидаты в Juniper, сеансы конфигурации в Arista и функция config replace в Cisco IOS.

По терминологии NAPALM управление полной конфигурацией в декларативном стиле определяется как операция замены конфигурации (configuration replace).

Для выполнения операций в более привычном режиме NAPALM также предлагает операцию объединения конфигураций (configuration merge), то есть возможность применения частичной конфигурации или выполнения лишь некоторых команд конфигурирования с проверкой их существования на целевом сетевом устройстве.

В любом случае, на основе операционной системы сетевого устройства изменения вступают в силу, только если они действительно необходимы. Ниже это будет продемонстрировано на нескольких примерах.

Начнем с выполнения операции замены конфигурации.

Выполнение операции замены конфигурации

Выполнение операции замены конфигурации означает, что на устройство передается полная активная конфигурация. Цель этой операции – обеспечить существование требуемой конкретной конфигурации на сетевом устройстве. По существу, мы объявляем (declare), какой должна быть конфигурация, не уделяя внимания командам типа no или delete.

Устройство Arista EOS с именем eos-spine1в текущий момент имеет следующую полную конфигурацию (информация о некоторых интерфейсах удалена в целях экономии места):

```
eos-spine1#show run
! Command: show running-config
! device: eos-spine1 (vEOS, EOS-4.15.2F)
!
! boot system flash:vEOS-lab.swi
!
transceiver qsfp default-mode 4x10G
!
hostname eos-spine1
```

Tlgm: @it_boooks


```
ip domain-name ntc.com
!
snmp-server community networktocode ro
!
spanning-tree mode mstp
!
aaa authorization exec default local
!
no aaa root
!
username ntc privilege 15 secret 5 $1$KergS3bl$RFVho/GXf.3bQHhOCbeky1
!
vrf definition MANAGEMENT
   rd 100:100
!
interface Ethernet1
   no switchport
!
interface Ethernet2
   no switchport
!
interface Ethernet3
   no switchport
!
interface Ethernet4
   no switchport
!
...
!
interface Management1
   vrf forwarding MANAGEMENT
   ip address 10.0.0.11/24
!
ip route vrf MANAGEMENT 0.0.0.0/0 10.0.0.2
!
ip routing
ip routing vrf MANAGEMENT
!
router ospf 100
   router-id 100.100.100.100
   network 10.0.0.10/32 area 0.0.0.0
   network 10.0.1.10/32 area 0.0.0.0
   network 10.0.2.10/32 area 0.0.0.0
   network 10.0.3.10/32 area 0.0.0.0
   network 10.0.4.10/32 area 0.0.0.0
   max-lsa 12000
!
management api http-commands
   protocol http
   no shutdown
   vrf MANAGEMENT
      no shutdown
```

Tlgm: @it_boooks


```
!
management ssh
   vrf MANAGEMENT
      no shutdown
!
!
end
eos-spine1#
```

Для выполнения операции замены конфигурации сначала необходимо сохранить конфигурацию, которую предполагается развернуть, локально на нашем сервере. Она сохраняется под именем *eos-spine1.conf* без внесения каких-либо изменений, то есть точно в том виде, в котором будет передана на устройство.

После этого конфигурация готова к развертыванию на целевом устройстве.

Прежде чем начать какие-либо действия с NAPALM, необходимо загрузить правильный драйвер и создать экземпляр объекта устройства NAPALM.



NAPALM можно быстро установить с помощью команды `pip install napalm`.

```
>>> from napalm import get_network_driver
>>>
>>> driver = get_network_driver('eos')
>>> device = driver('eos-spine1', 'ntc', 'ntc123')
>>>
```

Здесь `device` – переменная, содержащая объект устройства NAPALM. Этот объект имеет методы для работы с конфигурациями устройства, в том числе и для выполнения операции замены конфигурации. Операция выполняется с помощью метода `load_replace_candidate()`.

```
>>> device.open()   # требуется для загрузки регистрационных данных и установления
                      соединения
>>>                 # с устройством на основе используемого API
>>>
>>> device.load_replace_candidate(filename='eos-spine1.conf')
>>>
```

Единственной задачей метода `load_replace_candidate()` является загрузка конфигурации на устройство. Метод не вносит никаких изменений в текущую рабочую конфигурацию. Для Arista новая конфигурация загружается в активный сеанс. Этот процесс можно наблюдать визуально в интерфейсе командной строки EOS CLI:

```
eos-spine1#show configuration sessions
Maximum number of completed sessions: 1
Maximum number of pending sessions: 5

  Name             State         User    Terminal
  ---------------- ------------- ---------- --------
  napalm_574288    pending

eos-spine1#
```

Tlgm: @it_boooks



Драйвер Arista EOS NAPALM использует сеансы конфигурации для выполнения операций управления конфигурацией в среде NAPALM. Отметим, что способы выполнения методов в среде NAPALM в действительности являются различными в зависимости от конкретного драйвера устройства. Как уже было сказано ранее, Juniper использует конфигурации-кандидаты, Cisco IOS – функцию замены конфигурации, Cisco NXOS – checkpoint-файлы для полной замены конфигурации. Способы объединения конфигураций, которые будут рассматриваться ниже, также могут быть различными для каждого типа устройства.

В данном случае конфигурация загружается в активный сеанс EOS. Также можно воспользоваться подкомандой `diffs` для просмотра состояния или доступными в командной строке командами. Для EOS команда выглядит следующим образом: `show session-config named napalm_574288 diffs`.

Но более важным является универсальный метод NAPALM, возвращающий различия в командах, которые предполагается выполнить на целевом устройстве. Можно воспользоваться методом `compare_config()`, чтобы просмотреть команды, которые будут применены.

```
>>> diffs = device.compare_config()
>>> print(diffs)
>>>
```

Как и предполагалось, никаких отличий нет, поскольку была развернута та же конфигурация, которая существует на устройстве. Но если бы различия обнаружились, то можно было бы добавить дополнительную проверку средствами языка Python, затем выполнить передачу (коммит) конфигурации в активную работающую конфигурацию с помощью NAPALM-метода `commit_config()`.

```
>>> if diffs:
...     device.commit_config()
...
>>>
```

Далее подробно рассмотрим пример, в котором вносятся настоящие изменения в конфигурацию. На устройстве EOS в полном файле конфигурации существует единственная строка, определяющая сообщество:

```
!
snmp-server community networktocode ro
!
```

Мы удалим эту характеристику сообщества из конфигурации *eos-spine1.conf* и заменим ее двумя другими командами:

```
!
snmp-server community ntc ro
snmp-server community secret123 rw
!
```

Воспользуемся теми же двумя методами для загрузки конфигурации на устройство и для просмотра различий.

Tlgm: @it_boooks


```
>>> device.load_replace_candidate(filename='eos-spine1.conf')
>>>
>>> diffs = device.compare_config()
>>> print(diffs)
@@ -7,7 +7,8 @@
 hostname eos-spine1
 ip domain-name ntc.com
 !
-snmp-server community networktocode ro
+snmp-server community ntc ro
+snmp-server community secret123 rw
 !
 spanning-tree mode mstp
 !
>>>
```

Отметим, что теперь сгенерирован протокол различий и отмечены строки, которые существовали в старой конфигурации, и строки из нового конфигурационного файла. На устройство не передавались команды no. Вместо этого была развернута полная требуемая конфигурация, а EOS определила команды, которые необходимо удалить, и команды, которые необходимо добавить для применения новой конфигурации. Это весьма существенное различие, по сравнению с более привычными методиками управления конфигурацией.

Наконец, если различия в командах выглядят вполне корректно, можно применить (выполнить коммит) эти различия, то есть ввести их в действие с помощью метода `commit_config()`.

```
>>> device.commit_config()
>>>
```

Но если по каким-либо причинам необходимо вернуться к исходной конфигурации, то можно воспользоваться встроенным методом `rollback()`.

```
>>> device.rollback()
>>>
```

С точки зрения рабочего процесса (потока) и с учетом всех рассмотренных примеров мы вернулись к первоначальному состоянию с конфигурацией, в которой содержится одна строка определения сообщества SNMP.

В следующем разделе будет рассматриваться передача частичной конфигурации на коммутатор Arista.

Выполнение операции объединения конфигураций

Напомним, что для коммутатора Arista EOS в конфигурации существует только одна строка определения сообщества.

```
eos-spine1#show run | inc snmp-server
snmp-server community networktocode ro
eos-spine1#
```

Tlgm: @it_boooks

Создание (с помощью шаблонов Jinja) и развертывание полной конфигурации при каждом внесении изменений может быть затруднительным. В этом случае более разумно начать процесс автоматизации с управления конкретными характеристиками. В рассматриваемом примере нас интересует только характеристика SNMP. Следовательно, необходимо создать файл конфигурации с именем *snmp.conf*.

В файле *snmp.conf* размещены только две строки, определяющие характеристики сообщества, которые необходимо развернуть на целевом устройстве.

```
snmp-server community ntc ro
snmp-server community secret123 rw
```

Теперь все готово для развертывания требуемых команд из этого файла на целевом устройстве. Для этого воспользуемся методом `load_merge_candidate()`. Метод используется в тех случаях, когда нет необходимости передавать на устройство полную конфигурацию.

```
>>> device.load_merge_candidate(filename='snmp.conf')
>>>
>>> diffs = device.compare_config()
>>>
>>> print(diffs)
@@ -8,6 +8,8 @@
 ip domain-name ntc.com
 !
 snmp-server community networktocode ro
+snmp-server community ntc ro
+snmp-server community secret123 rw
 !
 spanning-tree mode mstp
 !
>>>
```

Обратите внимание на сгенерированный список различий. Отметим, что существующая характеристика сообщества SNMP не изменилась. Цель операции объединения конфигураций несколько другая. Она обеспечивает существование новой конфигурации (новых элементов конфигурации), но при этом не удаляет уже существующие конфигурации.

Операция объединения конфигураций не удаляет команды или какие-либо элементы иерархии конфигурации при использовании декларативной методики, но если вам известно, как работает NAPALM с конкретным драйвером устройства, то можно воспользоваться этим, чтобы получить определенные преимущества в управлении некоторыми функциональными возможностями в декларативном стиле.

Например, рассмотрим управление OSPF с использованием операции объединения конфигураций. Ранее уже демонстрировалась текущая конфигурация OSPF, которая выглядит следующим образом:

```
eos-spine1#show run section ospf
router ospf 100
```

Tlgm: @it_boooks


```
   router-id 100.100.100.100
   network 10.0.0.10/32 area 0.0.0.0
   network 10.0.1.10/32 area 0.0.0.0
   network 10.0.2.10/32 area 0.0.0.0
   network 10.0.3.10/32 area 0.0.0.0
   network 10.0.4.10/32 area 0.0.0.0
   max-lsa 12000
eos-spine1#
```

Создается файл конфигурации *ospf.conf*, в котором содержатся следующие команды:

```
router ospf 100
   router-id 100.100.100.100
   network 10.0.4.10/32 area 0.0.0.0
   network 10.0.5.10/32 area 0.0.0.0
   max-lsa 12000
```

Далее выполняется загрузка этих команд на целевое устройство.

```
>>> device.load_merge_candidate(filename='ospf.conf')
>>>
>>> diffs = device.compare_config()
>>>
>>> print(diffs)
@@ -54,6 +56,7 @@
    network 10.0.2.10/32 area 0.0.0.0
    network 10.0.3.10/32 area 0.0.0.0
    network 10.0.4.10/32 area 0.0.0.0
+   network 10.0.5.10/32 area 0.0.0.0
    max-lsa 12000
 !
 management api http-commands
>>>
```

Как можно было ожидать, здесь внесено только одно изменение: дополнительное определение новой подсети.

Но поскольку нам известно, что драйвер NAPALM для Arista EOS использует сеансы конфигурации (где все команды в сеансе применяются как единая транзакция), то можно воспользоваться этим преимуществом и декларативно управлять полной конфигурацией OSPF. Рассмотрим, как это делается.

Создается новая конфигурация OSPF с именем *ospf-2.conf*. Это та же конфигурация, которая использовалась ранее, но в нее просто добавляется команда `no router ospf 100` в самой первой строке файла.

```
no router ospf 100
router ospf 100
   router-id 100.100.100.100
   network 10.0.4.10/32 area 0.0.0.0
   network 10.0.5.10/32 area 0.0.0.0
   max-lsa 12000
```

Tlgm: @it_boooks

Загрузим эту конфигурацию OSPF на целевое устройство и посмотрим различия.

```
>>> device.load_merge_candidate(filename='ospf-2.conf')
>>>
>>> diffs = device.compare_config()
>>> print(diffs)
@@ -49,11 +51,8 @@
 !
 router ospf 100
    router-id 100.100.100.100
-   network 10.0.0.10/32 area 0.0.0.0
-   network 10.0.1.10/32 area 0.0.0.0
-   network 10.0.2.10/32 area 0.0.0.0
-   network 10.0.3.10/32 area 0.0.0.0
    network 10.0.4.10/32 area 0.0.0.0
+   network 10.0.5.10/32 area 0.0.0.0
    max-lsa 12000
 !
 management api http-commands
>>>
```

Здесь можно видеть, что окончательная версия конфигурации OSPF полностью совпадает с содержимым файла *ospf-2.conf*. Нет необходимости передавать несколько команд no для удаления ненужных определений подсетей. В этом рабочем потоке процесс OSPF не удалялся, а только лишь дополнялся. При правке характеристик OSPF не выполнялись операции удаления. Разумеется, вы сами можете проделать все требуемые операции, чтобы протестировать их работу на практике.

Получение данных от устройств с помощью NAPALM

Второй важной функцией, предоставляемой NAPALM, является возможность получения информации от сетевых устройств универсальным способом. Любые данные, полученные с помощью NAPALM, являются нормализованными и представлены в одинаковой форме для всех устройств, поддерживаемых NAPALM.

Напомним, что при рассмотрении различных API в главе 7 каждое устройство возвращало зависимые от конкретного производителя пары ключ-значение. Здесь есть один нюанс: даже если устройство поддерживает независимую от производителя модель данных, подобную моделям YANG от рабочей группы OpenConfig, упомянутой в главе 5, то следует учесть, что такая модель пока еще не получила широкой поддержки со стороны всех производителей сетевого оборудования.

У нас имеется ранее созданный экземпляр объекта устройства `device`. Воспользуемся встроенной функцией `dir()`, описанной в главе 4, чтобы получить список всех методов, которые поддерживает объект устройства NAPALM.

Tlgm: @it_boooks


```
>>> dir(device)
[...некоторые методы не показаны..., 'cli', 'close', 'commit_config', 'compare_config',
compliance_report', 'config_session', 'device', 'discard_config', 'enablepwd',
'get_arp_table', 'get_bgp_config', 'get_bgp_neighbors',
'get_bgp_neighbors_detail', 'get_config', 'get_environment', 'get_facts',
'get_firewall_policies', 'get_interfaces', 'get_interfaces_counters',
'get_interfaces_ip', 'get_lldp_neighbors', 'get_lldp_neighbors_detail',
'get_mac_address_table', 'get_network_instances', 'get_ntp_peers',
'get_ntp_servers', 'get_ntp_stats', 'get_optics', 'get_probes_config',
'get_probes_results', 'get_route_to', 'get_snmp_information', 'get_users',
'hostname', 'is_alive', 'load_merge_candidate', 'load_replace_candidate',
'load_template', 'locked', 'open', 'password', 'ping', 'port', 'profile',
'rollback', 'timeout', 'traceroute', 'transport', 'username']
>>>
```

Здесь можно видеть два метода, рассмотренных в предыдущем разделе, `load_merge_candidate()` и `load_replace_candidate()`, но, кроме того, следует отметить преобладающее количество методов `get_`, используемых для получения информации от сетевых устройств.

Рассмотрим некоторые из методов группы `get_` более подробно.

Первый метод, заслуживающий внимания, – `get_facts()`. Он используется для получения общей информации об устройстве, такой как ОС, время непрерывной работы, интерфейсы, производитель, модель, имя хоста и FQDN.

```
>>> device.get_facts()
{u'os_version': u'4.15.2F-2663444.4152F', u'uptime': 15645,
u'interface_list': [u'Ethernet1', u'Ethernet2', u'Ethernet3', u'Ethernet4',
u'Ethernet5', u'Ethernet6', u'Ethernet7', u'Management1'],
u'vendor': u'Arista', u'serial_number': u'', u'model': u'vEOS',
u'hostname': u'eos-spine1', u'fqdn': u'eos-spine1.ntc.com'}
>>>
```

Возвращаемые данные структурированы в универсальной форме вне зависимости от производителя запрашиваемого устройства, и это одно из главных преимуществ использования NAPALM. В приведенном примере NAPALM выполняет всю рутинную работу по нормализации, избавляя пользователя от необходимости выполнять операции по интеграции/преобразованию для устройств различных производителей, с которыми вы работаете. NAPALM всегда делает такую работу для пользователя.

Рассмотрим еще несколько примеров. Функция `get_snmp_information()` возвращает словарь, в котором собраны все данные о конфигурации SNMP на запрашиваемом устройстве:

```
>>> device.get_snmp_information()
{u'community': {u'networktocode': {u'mode': u'ro', u'acl': u''}}, u'contact': u'',
u'location': u'', u'chassis_id': u''}
>>>
```

Функция `get_lldp_neighbors()` возвращает словарь, содержащий список текущих известных («видимых») соседних устройств по протоколу LLDP с сортировкой их по отдельным интерфейсам:

Tlgm: @it_boooks


```
>>> device.get_lldp_neighbors()
{u'Ethernet2': [{u'hostname': u'eos-spine2.ntc.com', u'port': u'Ethernet2'}],
u'Ethernet3': [{u'hostname': u'eos-spine2.ntc.com', u'port': u'Ethernet3'}],
u'Ethernet1': [{u'hostname': u'eos-spine2.ntc.com', u'port': u'Ethernet1'}],
u'Ethernet4': [{u'hostname': u'eos-spine2.ntc.com', u'port': u'Ethernet4'}],
u'Management1': [{u'hostname': u'eos-spine2.ntc.com', u'port': u'Management1'},
{u'hostname': u'vmx2', u'port': u'fxp0'}, {u'hostname': u'vmx1', u'port': u'fxp0'},
{u'hostname': u'csr2.ntc.com', u'port': u'Gi1'}, {u'hostname': u'csr1.ntc.com',
u'port': u'Gi1'}]}
>>>
```

Обе упомянутые выше функции возвращают словарь. Можно создать небольшой скрипт для обработки и вывода словаря соседних устройств, определяемых по протоколу LLDP, в удобном для чтения формате:

```
>>> for interface, neighbors in device.get_lldp_neighbors().items():
...     print("INTERFACE: " + interface)
...     print("NEIGHBORS: ")
...     for neighbor in neighbors:
...         print("  - {}".format(neighbor['hostname']))
...
INTERFACE:  Ethernet2
NEIGHBORS:
  - eos-spine2.ntc.com
INTERFACE:  Ethernet3
NEIGHBORS:
  - eos-spine2.ntc.com
INTERFACE:  Ethernet1
NEIGHBORS:
  - eos-spine2.ntc.com
INTERFACE:  Ethernet4
NEIGHBORS:
  - eos-spine2.ntc.com
INTERFACE:  Management1
NEIGHBORS:
  - eos-spine2.ntc.com
  - vmx2
  - vmx1
  - csr2.ntc.com
  - csr1.ntc.com
>>>
```

Многие другие функции NAPALM работают похожим образом. Внимательное изучение формата вывода каждой из этих функций является правильным первым шагом в процессе их освоения, позволяющим выполнять более сложные задачи на основе текущего состояния сетевой среды.

Возможности интеграции NAPALM с другим ПО

NAPALM можно использовать для создания собственных специализированных приложений на языке Python. Ранее мы уже выяснили, что ядро NAPALM, по

Tlgm: @it_boooks

существу, представляет собой библиотеку, написанную на языке Python. Благодаря своей открытости и возможностям расширения, используемым в других Python-проектах с открытым исходным кодом, NAPALM весьма активно применяется в совокупности с другими инструментальными средствами, такими как Ansible, Salt и StackStorm.

Использование NAPALM в Ansible

Возможность интегрирования NAPALM в среду Ansible реализована в форме модулей Ansible. Эта реализация уже упоминалась в главе 9 в разделе, посвященном Ansible.

Существуют два основных модуля Ansible, соответствующих задачам, которые обсуждались ранее в этом приложении в разделе об управлении конфигурацией с помощью NAPALM и получении конфигурации устройств. Первый модуль называется `napalm_install_config` и используется для выполнения операций замены и объединения конфигураций. Второй модуль – `napalm_get_facts` – применяется как обертка для методов группы `get_`, поддерживаемых NAPALM.

Рассмотрим пример практического использования модуля `napalm_install_config`.

```
- name: DEPLOY CONFIGURATIONS WITH NAPALM
  napalm_install_config:
    hostname: "{{ inventory_hostname }}"
    username: "{{ un }}"
    password: "{{ pwd }}"
    dev_os: "{{ os }}"
    config_file: configs/snmp.conf
    diff_file: diffs/{{ inventory_hostname }}-snmp.diffs
    commit_changes: True
    replace_config: False
```

Модуль `napalm_install_config` поддерживает множество параметров. Предназначение некоторых параметров понятно из их имени, например `hostname`, `username` и `password`. Ниже приведено краткое описание других наиболее важных параметров:

- `dev_os` – ОС, драйвер устройства, управляемого NAPALM (например, `eos`, `ios`, `junos`);
- `config_file` – файл, содержащий команды конфигурации, которые должны быть загружены на сетевое устройство;
- `diff_file` – файл на сервере Ansible, содержащий протокол различий (diffs), сгенерированных с помощью метода `compare_config()`;
- `commit_changes` – это логическое (Boolean) значение. Если значение `True`, то будет выполнен `метод commit_config()`. Можно настроить так, чтобы команды не применялись, если необходимо только просмотреть различия;
- `replace_config` – это логическое (Boolean) значение. Если значение `True`, то будет выполнен метод `load_replace_candidate()`. Если значение `False`, то будет выполнен метод `load_merge_candidate()`.

Tlgm: @it_boooks

Если NAPALM-модули в среде Ansible установлены правильно, то можно также пользоваться утилитой `ansible-doc` для изучения дополнительных примеров практического применения модулей NAPALM:

```
ntc@ntc:~$ ansible-doc napalm_install_config
# вывод пропущен в целях экономии места
ntc@ntc:~$ ansible-doc napalm_get_facts
# вывод пропущен в целях экономии места
```



Как было отмечено ранее, в этом разделе не рассматриваются возможности интеграции NAPALM YANG. Для этой цели также существуют модули Ansible, но в нашей книге данная тема не обсуждается.

Использование NAPALM в Salt

Salt немного отличается от Ansible в плане интеграции с NAPALM, и эта особенность уже отмечалась в главе 9 при обсуждении применения NAPALM в среде Salt.

NAPALM интегрирован в Salt как штатный модуль, поэтому представляет собой наиболее часто используемый сетевой драйвер для управления сетевыми устройствами с помощью Salt.

Несколько вариантов интеграции более подробно описано в разделе Salt главы 9.

Существует специализированный вариант интеграции прокси-миньона для NAPALM. Ниже приведен пример соответствующей конфигурации:

```
proxy:
  proxytype: napalm
  driver: nxos
  fqdn: nxos-spine1.dc.amers
  username: ntc
  password: ntc123
```

Каждому методу NAPALM из группы `get_` соответствует один или несколько специализированных выполняемых модулей Salt, используемых для получения информации от устройств.

Пример получения статистических данных NTP от устройства с идентификатором миньона `vmx1`:

```
$ sudo salt vmx1 ntp.stats
# вывод пропущен в целях экономии места
```

Пример получения списка активных соседних устройств по протоколу BGP от устройства с идентификатором миньона `vmx1`:

```
$ sudo salt vmx1 bgp.neighbors
# вывод пропущен в целях экономии места
```

В Salt также имеется функция состояния `netconfig.managed`, которая выполняет операции замены и объединения конфигураций, рассмотренные в предыдущем разделе.

Tlgm: @it_boooks

Например, предположим, что существует SLS-файл состояния *ntp.sls*, содержащий следующие строки:

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - debug: true
```

Можно применить конфигурацию, сгенерированную из шаблона для этого устройства, используя интерфейс командной строки `salt`, но здесь главной задачей является просмотр выводимых данных, чтобы увидеть различия (diffs). Эти различия полностью совпадают с теми, которые можно было бы вывести с помощью NAPALM-метода `compare_config()`.

```
$ sudo salt vmx1 state.apply ntp
vmx1:
----------
          ID: ntp_peer_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 10:48:16.160777
    Duration: 4331.08 ms
     Changes:
              ----------
              diff:
                  [edit system ntp]
                  +    peer 10.10.10.1;
                  +    peer 10.10.10.3;
                  +    peer 10.10.10.2;
                  -    peer 1.2.3.4;
                  -    peer 5.6.7.8;
              loaded_config:
                  delete system ntp peer 1.2.3.4
                  delete system ntp peer 5.6.7.8
                  set system ntp peer 10.10.10.1
                  set system ntp peer 10.10.10.3
                  set system ntp peer 10.10.10.2

Summary for vmx1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   4.331 s
```

Использование NAPALM в StackStorm

В соответствующем разделе главы 9 было отмечено, что NAPALM также интегрирован в среду StackStorm. Интеграция NAPALM реализована в форме пакета (pack) StackStorm.

Tlgm: @it_boooks

Ниже приведен список операций NAPALM, поддерживаемых в пакете для StackStorm:

```
vagrant@st2vagrant:~$ st2 action list --pack=napalm -a ref
+-------------------------------------+
| ref                                 |
+-------------------------------------+
| napalm.bgp_prefix_exceeded_chain    |
| napalm.check_consistency            |
| napalm.cli                          |
| napalm.configuration_change_workflow |
| napalm.get_arp_table                |
| napalm.get_bgp_config               |
| napalm.get_bgp_neighbors            |
| napalm.get_bgp_neighbors_detail     |
| napalm.get_config                   |
| napalm.get_environment              |
| napalm.get_facts                    |
| napalm.get_firewall_policies        |
| napalm.get_interfaces               |
| napalm.get_lldp_neighbors           |
| napalm.get_log                      |
| napalm.get_mac_address_table        |
| napalm.get_network_instances        |
| napalm.get_ntp                      |
| napalm.get_optics                   |
| napalm.get_probes_config            |
| napalm.get_probes_results           |
| napalm.get_route_to                 |
| napalm.get_snmp_information         |
| napalm.interface_down_workflow      |
| napalm.loadconfig                   |
| napalm.ping                         |
| napalm.traceroute                   |
+-------------------------------------+
```

Здесь можно видеть прямые соответствия специализированным методам объекта устройства NAPALM, которые рассматривались выше в предыдущих разделах.

Пример получения общей информации (фактов) с использованием интерфейса командной строки StackStorm `st2`:

```
vagrant@st2vagrant:~$ st2 run napalm.get_facts hostname=vsrx01
# вывод пропущен в целях экономии места
```

Разумеется, полученные таким способом данные можно использовать в активных рабочих потоках StackStorm, как это было продемонстрировано в главе 9.

Tlgm: @it_boooks

Предметный указатель

Символы

\n, символ конца текущей строки и перехода на новую строку (EOL), 126
/, корневой каталог файловой системы Linux, 67
., обозначение текущего каталога в файловой системе Linux, 73
.., обозначение родительского каталога в файловой системе Linux, 70
#!, последовательность символов в начале скрипта bash, 80
~, символ домашнего каталога в файловой системе Linux, 68

A

Active Networks, 23
Ansible, 399
  Ansible Tower (Red Hat), 399
  inventory-файл, 401
  автоматизация серверов Linux, 400
  автоматизация сетевых устройств, 400
  атрибут задачи register, 426
  генерация отчета, 430
  генерация файла сетевой конфигурации, 418
  запись данных в файл, 426
  использование NAPALM, 594
  использование для автоматизации сети, 414
  команда show, 426
  комплект сценариев (playbook), 408
  модуль, 410
    eos_config, 421
    file, 419
    ios_command, 410
    NAPALM, 435
    napalm_get_facts, 594
    napalm_install_config, 594
    NTC, 434
    template, 418
    независимых авторов, 433
    отладки debug, 425
    параметр commands\, 411
    параметр provider\, 411
    тип, 415
    тип config, 421
    тип facts, 424
  ограниченный режим (limit), 422
  определение, 399
  основы работы, 400
  проверка согласованности (compliance check), 428
  режим подробного вывода (verbosity), 422
  режим проверки (check mode), 422
  сценарий, задача (task), 410
  утилита ansible-doc, 415
  файл переменных, 412, 416
  шаблон Jinja, 417
  шаблон конфигурации, 415
Arista Networks, компания, 31
ASIC, application-specific integrated circuits. *См.* Интегральная схема специального назначения

B

Bare-metal switching. *См.* Аппаратная коммутация
BGP, Border Gateway Protocol. *См.* Протокол граничного шлюза (BGP)
Big Switch Networks, компания, 23, 33

Tlgm: @it_boooks

Big Switch, компания, 60
BitKeeper, 329
BPDU, Bridge Protocol Data Units, 108

C

Canonical Ltd., компания, 65
CI pipeline. *См.* Непрерывная интеграция: конвейер
Cisco
  ASA RESTful API, 245
  IOS-XE
    изменение конфигурации с помощью библиотеки ncclient, 306
    использование библиотеки ncclient, 294
    сетевой API RESTCONF, 282
  IOS-XR, изменение конфигурации с помощью библиотеки ncclient, 313
  Nexus NX-API, 265
Cisco Application Virtual Switch (AVS), 29
Cisco IWAN, компания, 36
Cisco Nexus 1000V, 29
CloudGenix, компания, 36
Commodity switching, 33
Continuous Delivery. *См.* Непрерывная доставка
Continuous Deployment. *См.* Непрерывное развертывание
Continuous Integration. *См.* Непрерывная интеграция
Cumulus Networks, компания, 33, 60
Система управления IP-адресами (IP address management system, IPAM), 45
Система управления сетью (network management system, NMS), 45

D

Data center network fabric. *См.* Сетевая фабрика центров данных
Debian, формат пакетов deb, 65
DevOps, 548
Django, 210
Docker, 566, 568
DVCS, distributed version control system. *См.* Система управления версиями: распределенная

F

FIB, Forwarding Information Base. *См.* Информационная база данных переадресации (FIB)
ForCES, Forwarding and Control Element Separation, 23

G

Genshi, 211
Gerrit, 515
  change (изменение), 516
  patchset (набор патчей), 516
Git, 329
  HEAD, указатель, 343
  архитектура, 331
  ветвь, 358
    выбор активной (checkout) ветви, 364
    объединение, 366
    определение, 358
    создание, 363
    удаление, 366, 368
  внесение информации о пользователе, 335
  главные цели, 329
  индекс, 331
  история создания и развития, 329
  команда
    git add, 334, 336, 339, 343
    git branch, 363
    git branch -a, 374
    git branch -d, 368
    git cat-file, 351
    git checkout, 364
    git clone, 377
    git clone --bare, 384
    git commit, 337, 338, 339, 343
    git commit -a, 341

Tlgm: @it_boooks

git commit -m, 342
git config, 336
git diff, 354
git fetch, 375
git fetch --prune, 388
git help, 342
git init, 333
git init --bare, 383
git log, 337, 348, 349
git log --oneline, 348, 350, 372
git ls-tree, 348, 352
git merge, 367, 376
git pull, 376
git push, 386
git remote, 373
git remote add, 373
git remote update, 374, 375
git remote -v, 382
git reset, 344
git status, 334, 336, 340, 343, 347
коммит, 331
внесение изменений, 336
в отслеживаемые файлы, 339
выполнение, 335
изменение (редактирование), 339
корректировка, 338
рекомендации по выполнению, 337
контрольная точка (checkpoint), 337
Механизм remote, 373
объект коммит (commit object), 337
определение различий между версиями файлов, 354
отмена фиксации файла в индексе, 343
получение подробной информации о конкретном коммите, 351
рабочий каталог, 330
репозиторий, 330, 333
глобальное исключение файлов, 349
добавление файлов, 333
запрос на включение изменений (pull request), 393
исключение файла, 345
клонирование, 377
неполный (bare repository), 382
определение, 331
ответвление (forking), 390
просмотр подробной информации, 349
совместно используемый, 382
совместно используемый, ввод изменений, 386
создание, 333
совместная работа группы пользователей, 371
совместная работа с онлайновми сервисами, 389
специальный файл .gitignore, 347
терминология, 330
указатель (объект) remote, 373
установка, 332
файл .gitignore_global, 349
хранение секретной информации, 346
GitHub, 390, 514
pull request (запрос на принятие изменений), 516
запрос на включение изменений (pull request), 393
ответвление репозитория, 390
GitLab, 515, 539
merge request (запрос на объединение ветвей), 516
Glue Networks, компания, 36
GNU General Public License (GPL), лицензия, 61
GNU Project, 61
Go, 211

H

HTTP
заголовок
Accept, 247
Content-Type, 247
код ответа, 234
тип запроса, 233

Tlgm: @it_boooks

DELETE, 233
GET, 233
PATCH, 233
POST, 233
PUT, 233

I

IPAM, IP address management, 444

J

Jenkins, сервер сборки ПО с открытым исходным кодом, 520
Jinja, 213
библиотека Jinja2, 215
динамическая вставка данных в шаблон, пример, 214
документ Jinja FAQ, 218
команда
block, 227
include, 227
конфигурация для одного порта коммутатора, пример, 218
конфигурация нескольких портов коммутатора, пример, 219
наследование шаблона, 227
обработка шаблона средствами языка Python, 215
определение, 213
преимущества, 213
создание переменной в шаблоне, 228
условное выражение, 217
фильтр, 224
функция render(), 216
цикл, 217
шаблон Ansible, 417
JSON, JavaScript Object Notation. *См.* Формат данных: JSON
Juniper Contrail, 29

L

Libvirt, виртуальная сеть, 578
Libvirt, API виртуализации, 553, 554
Linux
Ansible, 60
bash
shebang (#!), 80
промпт (prompt), 68
путь поиска (search path), 73, 80
путь поиска, настройка, 82
Cumulus Linux, 60
init-скрипт, 82
Open Network Linux (ONL), 60
Open vSwitch. *См.* OVS
Switch Light, 60
systemd, 83
команда systemctl, 83
Vagrant, инструментальное средство для создания рабочей среды, 69
виртуальная машина в сетевой среде, 553
виртуальная машина в сети, шлюз, 554
выполнение программ, 79
выполняемый (executable) файл, 79
право на выполнение, 79
скрипт на языке командной оболочки bash, 79
демон (daemon), 82
CentOS, 86
Debian, 83
Ubuntu, 84
список установленных сетевых соединений, 86
дистрибутив, 62
CentOS, 63
Debian, 64
Fedora, 63
Red Hat Enterprise Linux (RHEL), 62
Ubuntu, 65
интерфейс ipvlan, 569
интерфейс macvlan, 551
интерфейс macvtap, 557
команда
brctl, 107
cd, 69

Tlgm: @it_boooks

cd без параметров, возврат в домашний каталог, 71
cd -, возврат в предыдущий каталог, 71
chgrp, 79
chmod, 78
chown, 79
cp, 75
echo $PATH, 80
file, 79
ip, 87, 95
ip link, 107
ip link, размещение интерфейса в сетевом пространстве имен, 561
ip route, 98
ip, соединение сетевых пространств имен с помощью пары veth, 565
ip, создание интерфейса macvlan, 551
ip, создание сетевого пространства имен, 560
ls, 77
man, 75
mkdir, 73
mv, 75
ps, 86
pwd, 69
rm, 74
service, 82
ss, 86
touch, 72
which, 81
командная оболочка shell, 66
bash, Bourne Again Shell, 66
коммутатор, 105
практический пример, 105
коммутация, 105
контейнер, 566
LXC (LinuX Containers), 567
краткая история создания, 61
маршрутизация
BIRD, 104
Quagga, 104
для конечного хоста, 98
конфигурация маршрутизатора (router), 103
протокол динамической маршрутизации, 104
механизм групп управления (control groups — cgroups), 566
пара veth, 564
переменная среды (environment variable) PATH, 82
права доступа
владельца, 76
выполнение, 76, 77
группы, 76
для всех, 76
запись, 76
на совершение действий, 76
пользователя, 76
рекомендации по изменению, 78
чтение, 76
проблема переносимости пакетов между дистрибутивами, 64, 65
процесс, 86
работа с файлами и каталогами, 72
изменение прав доступа, 76
копирование, 75
копирование каталогов, особенности, 75
переименование, 75
перемещение, 75
создание, 72
удаление, 74
сервис (service), 82
сетевое пространство имен, 558
сетевой интерфейс, 87
IP-адрес, 91
VLAN, 95
VLAN, команда ip, 95
VLAN, пример конфигурирования, 96
значение MTU, 90
команда ip для конфигурирования, 88

Tlgm: @it_boooks

конфигурирование в командной строке, 87
отдельный файл конфигурации для каждого интерфейса, 94
отдельный файл конфигурации, рекомендации, 104
пакет iproute2, 87
тип, 87
файл конфигурации, 92
символическая ссылка (symbolic link) на файл или каталог, 85
система инициализации (init system), 82
специальная команда ip netns exec для выполнения в сетевом пространстве имен, 562
стартовый (startup) скрипт, 82
файловая система
абсолютный путь (absolute path), 70
домашний каталог (home directory) пользователя, 68
каталог, 68
навигация, 67
один корень (single-root filesystem), 67
относительный путь (relative path), 70
путь (path), 68
родительский каталог (..), 70
способы перемещения, 71
текущий каталог (current directory) (.), 73
характеристика и преимущества, 60
шлюз, 554

M

Mako, 211

N

NAPALM, 583
возможности интеграции с другим ПО, 593
документация, 583
драйвер Arista EOS, 587
использование библиотеки netmiko, 322
использование в Ansible, 594
использование в Salt, 595
использование в StackStorm, 596
метод
commit_config(), 587, 588
compare_config(), 587, 596
get_facts(), 592
get_lldp_neighbors(), 592
get_snmp_information(), 592
load_merge_candidate(), 589, 592
load_replace_candidate(), 586, 592
rollback(), 588
модуль для Ansible, 435
операция замены конфигурации (configuration replace), 584
выполнение, 584
операция объединения конфигураций (configuration merge), 584
выполнение, 588
основные функции, 583
получение данных от сетевых устройств, 591
управление сетевой конфигурацией, 583
управление сетевой конфигурацией в декларативном стиле, 584
установка, 586
функция dir(), 591
NETCONF, 55
описание и характеристики, 55
Network to Code, компания, 528
NFV, Network Functions Virtualization. *См.* Виртуализация сетевых функций
NOS, network operating system. *См.* Сетевая операционная система
Nuage Virtual Service Platform (VSP), 29

Tlgm: @it_boooks

O

OpenCompute Project (OCP), рабочая группа, 60
OpenDaylight (ODL), специализированный SDN-контроллер, 38
OpenFlow, 21, 26
Open vSwitch. *См.* OVS
Open vSwitch (OVS), 29
OVS, 570
  использование виртуальных машин, 578
  использование внутренних портов, 579
  использование гипервизора KVM, 578
  использование контейнеров, 577
    Docker, 577
    LXC, 577
  конфигурирование, 572
  соединение нескольких типов рабочих нагрузок, 575
  соединение сетевых пространств имен, 575
  установка, 570
  характеристика и преимущества, 570

P

PAM, Pluggable Authentication Modules, тип аутентификации, 476
PBR, Policy Based Routing. *См.* Маршрутизация на основе правил
PCE, Path Computation Element, 23
Pica8, компания, 33
Postman, 247, 306
Publisher ACL, 476
Python
  shell, командная оболочка, 115
  доступ к некоторым частям строки или элементов в списке, 173
  интерактивный интерпретатор, 115
  использование RESTCONF API, 289
  использование формата данных YAML, 188
  команда
    from import as, для модулей, 168
    from import, для модулей, 168
    import, для модулей, 167
  комментарий (символ #), 175
  комплексный пример использования условных выражений, составных объектов и циклов, 151
  краткое введение в основы практического применения классов, 177
  метод
    append(), список, 135, 153
    count(), список, 136
    count(), строка, 124
    endswith(), строка, 122
    format(), строка, 125, 152
    get(), словарь, 140, 153, 161
    index(), список, 137
    insert(), список, 135
    isdigit(), строка, 123
    items(), словарь, 142, 152
    join(), строка, 126
    keys(), словарь, 141
    lower(), строка, 121
    pop(), словарь, 141
    pop(), список, 136
    readlines(), файл, 159
    read(), файл, 159
    sort(), список, 137
    splitlines(), строка, 160
    split(), строка, список, 127
    startswith(), строка, 121
    strip(), строка, 123, 160
    update(), словарь, 142
    upper(), строка, 121
    values(), 141
    write(), файл, 161
    сравнение с функцией, 134
  модуль, 167

Tlgm: @it_boooks

импортирование, 167
определение, 167
наиболее удачный выбор для начального обучения, 113
обработка шаблона Jinja, 215
объект, термин containment (способность объекта содержать другие объекты), 148
оператор
elif, 146
else, 147
else if, 146
if, 146
if, символ двоеточия (:), 146
in, 148
математический, 128
!= (не равно), 132
== (равно), 132
операция
вычисление остатка от деления (или деление по модулю) (%), 130
вычитание (-), 129
деление (/), 129
сложение (+), 128
умножение (*), 129
операция with (менеджер контекста, автоматически закрывает файл), 162
операция объединения (конкатенации) строк, 119
пакет
pip, программа установки, 171
зависимости при установке, 172
обновление, программа pip, 171
репозиторий Python Package Index, PyPI, 171
установка, 171
установка из исходных кодов, 172
установка, пример установки пакета библиотеки netmiko, 171
получение нескольких первых или нескольких последних символов строки или элементов списка, 173
преимущества изучения для сетевого инженера, 113
преобразование строки в целое число, 173
преобразование целого числа в строку, 173
при запуске скрипта флаг -i позволяет войти в интерпретатор, 174
проверка объектов (пустые/не пустые), 175
проверка типа переменной, 174
программа
выполнение, 165
создание, 163
работа с файлами, 158
запись, 161
менеджер контекста with, 162
операция close (закрыть), 161
операция open (открыть), 159
режим доступа, 163
чтение, 158
чтение, пример, 158
работа с форматом данных JSON, 200
синтаксический разбор (парсинг) содержимого объектов разнообразных типов, 173
скрипт
выполнение, 165
выражение if __name__ == \, 165
комбинация символов shebang (#!), 164
передача аргументов из командной строки, 169
перемещение кода из интерпретатора, 166
создание, 163
создание, пример, 163
список аргументов командной строки sys.argv, 170
смещение в 4 пробела для блока кода (соглашение по стилю оформления), 146

Tlgm: @it_boooks

совместное использование функций dir(), type(), help(), 173
способ формирования строк с использованием специального символа %, 175
строка
  применение оператора умножения, 129
  сравнение без учета регистра букв, 121
  форматирование, 125
строка создания документации (docstring), 176
тип данных, 117
  встроенный метод, 117
  кортеж, 143
  кортеж, неизменяемая (immutable) структура, 144
  кортеж, сравнение со списком и множеством, 145
  логический, 130
  логический, False, 130
  логический, True, 130
  логический, инвертирование, 132
  логический, таблица вычисления истинных и ложных значений, 130
  множество, 143
  множество, создание и использование, пример, 144
  множество, сравнение со списком и кортежем, 145
  словарь, 138
  словарь, выражение dict[key], 139
  словарь, итерация, 142
  словарь, обновление, 142
  словарь, определение, 138
  словарь, ошибка KeyError (ключ не существует), 140
  словарь, пара ключ-значение (key-value), 139
  словарь, преобразование из списка, 139
  словарь, создание, пример, 139
  словарь, удаление элемента, 141
  словарь, цикл прохода, 142
  словарь, элемент (item), 139
  список, 133
  список, вставка элемента, 135
  список, вывод элемента, 134
  список, добавление элемента, 135
  список, индекс (index) элемента, 134
  список, нумерация элементов и получение значения индекса, 154
  список, преобразование в словарь, 139
  список, создание, 133
  список, создание пустого списка, 135
  список, сортировка, 137
  список, сравнение со множеством и кортежем, 145
  список, удаление элемента, 136
  строка (string), 118
  целое число (integer), 128
условное логическое выражение, 145
  комбинация if-elif-else, 147
  оператор if, 145
  ошибка синтаксиса, 147
функция, 154
  dir(), 119, 134
  enumerate(), 154
  help(), 120
  isinstance(), 174
  len(), 134
  type(), 119, 128
  передача переменных (параметров), 155
  пример, 155
  сравнение с методом, 134
цикл, 149
  for, 150
  for-each, 150
  for-in, 150
  while, 149

Tlgm: @it_boooks

R

RCP, Routing Control Platform, 23
REST, 232
  архитектурные ограничения, 232
  определение, 56
RESTful, 232
RESTful, характеристика, 56
RPC, remote procedure call.
См. Удаленный вызов процедур (RPC)
RPM Manager, менеджер пакетов
RPM, 64
Ruby, 211

S

SaaS, Software as a Service — программное обеспечение как услуга, 514
Salt, 437
  grains, 448
  master, 438
  minion, 438
  pillar, 443
  RESTful API, 468
  Thorium, 474
  top-файл, 444
  автоматизация сетевых устройств, 440
  архитектура, 437
  выполнение функций в удаленном режиме, 467
  выполняемый модуль (execution module), 448
  генерация отчетов, 466
  главный файл конфигурации (master configuration file), 456
  журнал (log), 477
  интерфейс SDB (small database queries), 476
  инфраструктура, управляемая событиями (шина или канал событий — event bus), 469
  использование NAPALM, 595
  использование библиотеки pepper, 469
  комплексное соответствие (compound matching), 451
  кеш, 476
  маяк (beacon), 470
  механизм указания цели (targeting), 451
  модуль
    test, 452
    ключ --out, вывод в заданном формате, 453
    расширения, 477
    состояний (states), 455
    состояния для автоматизации сети, 455
  опция sys.doc, 452
  пакет salt-ssh, 438
  передача данных во внешний сервис, 454
  планировщик (scheduler), 454
  прокси-миньон (proxy minion), 439
  реактор (reactor), прослушивание шины событий, 472
  ретернер (returner), 466
  сбор данных об устройствах, 450
  система Publisher ACL, 476
  состояние SLS-файла (state SLS), 455
  специализированный механизм (engine) для перенаправления событий, 471
  управление сетевыми конфигурациями, 458
  формат файла SLS, 441
  эффективные практические методики, 475
SDB, Small database queries, 476
SDN, Software Defined Networking.
См. Программно определяемая сеть
SD-WAN, Software Defined Wide Area Networking. *См.* Программно определяемая глобальная сеть (SD-WAN)
Silverpeak, компания, 36
Snap, инструментальная рабочая среда, 532

Tlgm: @it_boooks

SNMP, протокол, 46, 53, 532
Source control. *См.* Система управления исходным кодом
SSH, 54
StackStorm, 479
  архитектура, 482
  документация, 484
  использование NAPALM, 596
  конфигурирование входящих обратных HTTP-вызовов webhooks, 494
  операция (action), 480, 484
  основы, 480
  пакет napalm, пример использования, 486
  пакет (pack), 482
  правило (rule), 481, 496
  рабочий поток (workflow), 484
  сенсор (sensor), 493
  триггер (trigger), 494
STP, Spanning Tree Protocol. *См.* Протокол остовного дерева (STP)

T

TAP, устройство, 555
Telnet, 54
Test-Driven Development, TDD, 508
Thorium, 473
ToDD, инструментальное средство тестирования, 532
Tox, 524

V

Vagrant, 525
  файл Vagrantfile, 525
VeloCloud, компания, 36
Viptela, компания, 36
Virtual eXtensible LAN (VxLAN), 29
Virtual Routing and Forwarding (VRF), логическая сеть, 37
VLAN, логический (виртуальный) сетевой интерфейс, 95
VMware distributed switch (VDS), 29
VMware NSX, 29
VMware standard switch (VSS), 29
VPN-туннель, 36

W

Webhook, обратный HTTP-вызов, 494
White-box switching. *См.* Коммутация с помощью прозрачного ящика

X

XML. *См.* Формат данных: XML, представление данных Junos, пример, 182
XQuery, 197
XSD, XML Schema Definition, 192
XSLT (Extensible Stylesheet Language Transformations), 194

Y

YAML. *См.* Формат данных: YAML
  двойные фигурные скобки, 45
YANG, 202
  документ RFC 6020, 202
  команда
    container, 205
    leaf, 204
    leaf-list, 204
    list, 205
  обзор, 202
  определение, 202
  тип модели, 203
  язык моделирования данных, 202
  язык моделирования, специально предназначенный для сетевой среды, 207

Z

ZeroMQ, 469

А

Автоматизация сети, 32
  Salt, 437, 440
  StackStorm, 479
  важность предварительного тестирования, 510
  влияние концепции открытых сетей, 57

Tlgm: @it_boooks

инструментальное средство
Ansible, 399
обзор, 396
использование Ansible, 414
качество системы, 510
конвейер непрерывной интеграции, 512
методика создания шаблонов сетевой конфигурации, 44
непрерывная интеграция, 500
непрерывная интеграция и непрерывная доставка, 507
организационная стратегия, 536
важность поддержки со стороны руководства, 538
восприятие критических сбоев (ситуаций), 541
парадигма «создать или купить», 540
преобразование организации старого образца, 536
переход между платформами, 47
платформа On Demand Labs, 528
подготовка и настройка устройств, 43
Ansible, 44
Salt, 44
пример, 44
файл переменных на языке YAML, пример, 44
шаблон Jinja, пример, 45
по методике разработки через тестирование, 531
практические навыки и обучение (рекомендации), 543
важность глубокого изучения основных принципов, 545
ИТ-сертификация, 546
преимущества системы управления исходным кодом, 328
преимущество простых решений и структур, 502
проблема организации рабочего процесса, 503
программно управляемая сеть (SDN), 58
рекомендации и предпосылки, 502
рекомендация, 43
роль человека, 501, 547
сбор данных, 46
модель активной передачи данных (push model), 47
потоковая телеметрия (streaming telemetry), 47
пример, 46
составление отчета, 50
сбор подробных телеметрических данных, 532
с использованием сетевых API, 261
создание и развитие культуры, 535
тип, 43
управление конфигурацией, 49
развитие от SNMP до API устройств, 53
совместимость, 49
устранение проблем и неисправностей, 51
цели и преимущества, 40
шаблон
правила использования, 229
преимущества, 212
преимущество, 208
Автоматическая установка и конфигурирование (zero-touch provisioning, ZTP), 37
Андерлейная (нижележащая) сеть (underlay network), 30
Аппаратная коммутация, 33
Архитектура «колеса» (центральная ось и спицы — hub-and-spoke architecture), 438

Б

База данных с управляющей информацией (management information base, MIB), 46
База данных управляющей информации (management information bases, MIB), 54

Tlgm: @it_boooks

Библиотека
Libvirt, 578
ncclient, 262, 292
NETCONF-операция delete, выполнение, 308
NETCONF-операция replace, выполнение, 309
атрибут ответа NETCONF, примеры, 298
вызов методов NETCONF API, 292
выполнение операций, специализированных для конкретных производителей оборудования, 314
изменение конфигурации Cisco IOS-XE, 306
изменение конфигурации Cisco IOS-XR, 313
изменение конфигурации Juniper vMX Junos, примеры, 310
метод get(), 294
модуль manager, 292
объект Manager, 293
ответ NETCONF, содержимое, 296
получение конфигурации Cisco IOS-XE, 294
получение конфигурации Juniper vMX Junos, 303
установка, 292
фильтр для минимизации данных ответа, 300
netmiko, 46, 262, 317
импорт объекта устройства, 317
использование в NAPALM, 322
команды, передаваемые на устройство, 317
передача команд на устройство, 319
переход в режим конфигурации, 319
полная идентичность использования на устройствах различных производителей, 317
преимущества, 317
проверка доступных методов, 318
проверка промпта устройства, 318
управление сетевым устройством через SSH-соединение, 317
установка, 317
установление SSH-соединения, 318
pepper, 469
pyeapi, 282
requests, 262
запрос GET, пример, 262
использование, 262
обновление описания интерфейса, 264
передача данных в теле HTTP-запроса, 265
установка, 262
Блоб (blob — binary large object), 337

В

Виртуализация сетевых функций, 26
Виртуализация сети, 29
Виртуальная коммутация, 29
Виртуальная маршрутизация и переадресация (Virtual Routing and Forwarding, VRF), 559
Виртуальная машина, 553
гипервизор KVM, 553, 578
шлюз, 554

Г

Глобальная сеть (WAN), 36

Д

Декларативная конфигурация, 584

З

Зависимость (dependency) пакета ПО, определение, 63
Запрос на включение изменений (pull request), 394

И

Идемпотентность (idempotency), 420
Инструментальное средство развертывания, 528

Tlgm: @it_boooks

Инструментальное средство тестирования, 531
Интегральная схема специального назначения, 21
Интерфейс
  ipvlan, 569
  macvlan, 551
    конфигурирование, 551
    практическое использование, варианты, 551
    создание, команда ip, 551
    сравнение с интерфейсом VLAN, 550
    удаление, 551
  macvtap, 557
Интерфейс командной строки (CLI), 54
Информационная база данных переадресации (FIB), 22
Инфраструктура как код (Infrastructure as Code — IaC), методика, 480
ИТ-сертификация, 546

K

Касадо, Мартин (Martin Casado), 20, 23
Комиссия по утверждению изменений (Change Approval Board – CAB), 514
Коммутация с помощью прозрачного ящика, 33

M

МакКеон, Ник (Nick McKeown), 21
Маршрутизация на основе правил (PBR), 22
Мгновенная реакция на критический отказ (failing fast), 541
Мердок, Иэн (Ian Murdock), 64
Механизм коммутации MPLS, 36
Модель данных, 180
  Kwalify, инструментальное средство описания для YAML, 189
  в JSON (JSON Schema), 201
  в XSD (XML), 192
  важные концепции, 202
  в формате YAML, 188
  выбор языка, рекомендации, 206
  определение, 189
  язык YANG, 202
Модель клиент-сервер без сохранения состояния (stateless client-server model), 57
Модель «оплата по мере роста» (pay-as-you-grow model), 27

H

Непрерывная доставка, 506
Непрерывная интеграция, 501
  защита от человеческих ошибок, 501
  конвейер, 505, 512
    автоматизация сборки (build automation), 519
    автоматизация сборки (build automation), пример, 521
    инструментальное средство развертывания, 528
    использование инструментальных средств тестирования для автоматизации, 531
    использование системы управления исходным кодом, 513
    методика развертывания ПО в контейнерах Docker, 529
    основные компоненты, 512
    пример использования GitLab, 515
    процедура отката (rollback), 508
    рецензирование коллегами, 513
    среда с поэтапной организацией, 507
    среда тестирования, разработки и организации этапов, 524
  обзор методики, 504
  основные принципы, 502
  основы, 504

Tlgm: @it_boooks

применимость и преимущества для сетевой среды, 511
разработка через тестирование, 508
релиз-инженер (release engineer), 506
участие человека, 505
фрагмент кода с исправлением («заплатка» — patch), 513
цель, 504
улучшение надежности, 504
ускоренная реакция, 504
Непрерывное развертывание, 504, 507

О

Обработка критических ситуаций (embracing failure), 541
Оверлейная сеть (overlay network), 30
Открытая сеть, концепция, 57

П

Пара veth, виртуальный Ethernet (Virtual Ethernet), 564
Парадигма «создать или купить», 541
Перенаправление (forwarding) пакетов, 21
Проблема накопления «технических задолженностей» (technical debt), 509
Программно определяемая глобальная сеть (SD-WAN), 36
Программно определяемая сеть
возникновение технологии, 20
определение, 25
технологии и направления, 25
Программно управляемая сеть (SDN), автоматизация сети, 58
Протокол граничного шлюза (BGP), 32
Протокол остовного дерева (STP), 109
Протокол ретрансляции кадров Frame Relay, 36

Р

Рабочая нагрузка (workload), 575
Разработка через тестирование, 508
Рецензирование коллегами, 513

С

Сетевая автоматизация, артефакты, 325
Сетевая конфигурация
внесения изменений в форме транзакции (transaction), 237
генерация файла в Ansible, 418
декларативная (declarative coniguration), 290
первоначальная (startup configuration), 236
текущая рабочая (running configuration), 236
управление с помощью NAPALM, 583
управление с помощью Salt, 458
хранилище данных, конфигурация-кандидат (candidate configuration), 236
шаблон, 208
Ansible, 415
наследование, 227
правила использования, 229
преимущества, 212
приложение (язык), 209
требования к языку, 209
три этапа работы (создание, форма данных, управление данными), 209
язык Jinja, 213
Сетевая операционная система, 26
Сетевая операционная система (NOS), 60
Сетевая фабрика центров данных, 35
Big Switch Big Cloud Fabric (BCF), 36
Cisco Application Centric Infrastructure (ACI), 36
фабрика и гиперконвергентная сеть (hyper-converged network) компании Plexxi, 36
Сетевое пространство имен
в Linux, 558
выполнение команды, 562

Tlgm: @it_boooks

практическое использование, варианты, 559
размещение интерфейса, 560
соединение с помощью пары veth, 565
создание, 560
удаление, 560
Сетевое устройство
прикладной программный интерфейс (API), 31
уровень представления данных, 21, 26
уровень управления, 21, 26
Сетевой API, 53
Arista eAPI, 275
eAPI, 31
NETCONF, 55, 236
документ RFC 6241, 236
идентификатор сеанса (session ID), 252
интерактивный сеанс, 252
операция, 239
операция delete, выполнение с помощью библиотеки ncclient, 308
операция <edit-config>, 240
операция <get>, 239
операция replace, выполнение с помощью библиотеки ncclient, 309
операция, список, 242
определение, 236
основы использования, 323
правильный способ формирования XML-запроса на внесение изменений, 259
практическое применение, 252
прерывание интерактивного сеанса (Ctrl-C), 254
символы]]>]]>, обозначение завершения запроса, 254
сообщение (message), 238
стек протоколов, 237
транспортный протокол (SSH), 238
уровень содержимого или контента (content), 243
формирование правильного запроса, 252
функциональные характеристики (возможности), 243
хранилище данных конфигурации-кандидата (candidate configuration), 236
Nexus NX-API, 31
RESTCONF, 282
активизация, 282
вызов метода, 283
запрос PATCH, 287
запрос PUT, 287
использование языка Python, 289
ответ, пример, 286
поддержка протокола HTTP, 283
сравнение с RESTful API, 283
RESTful, 231
основы, 231
RESTful API, 57
SSH, 54
Telnet, 54
использование для автоматизации сети, 261
на основе NETCONF, 231
на основе протокола HTTP, 231, 234
cURL, 245
cURL, взаимодействие с интерфейсом Cisco ASA RESTful API, 245
взаимодействие с Postman, 247
не использующий концепции RESTful, 235
основы использования, 323
практическое применение, 245
обзор, 53
протокол SNMP, 53
тип, 231
Сетевой протокол, 181
Система управления версиями распределенная, 347

Tlgm: @it_boooks

Система управления версиями (version control), 326
Система управления доступом Role Based Access Control (RBAC), 399
Система управления изменениями (revision control), 326
Система управления исходным кодом, 325, 513
  Git, 329
  варианты (примеры) использования, 326
  ответственность за внесение изменений (учетная запись), 327
  отслеживание изменений, 326, 327
  отслеживание состояния файлов, 326
  преимущества, 326
  создание правильного процесса и рабочего потока (workflow), 327
Специализированный сетевой контроллер, 38
Столлман, Ричард (Richard Stallman), 61

Т

Тестирование обработки критического сбоя (failover testing), 533
Торвальдс, Линус (Linus Torvalds), 61, 329

У

Удаленный вызов процедур (RPC), 238
Узел/система сетевого управления (network management station, NMS), 53
Универсальный многопротокольный механизм передачи данных – multiprotocol label switching, MPLS, 36

Ф

Филдинг, Рой (Roy Fielding), 232
Формат данных, 180
  JSON, 197
    JSON Schema, для описания модели данных, 202
    использование в коде Python, 200
    как упрощенная версия XML, 198
    основы, 198
    фигурные скобки {} для формирования блока данных (объекта), 199
  XML, 190
    XML Schema Definition (XSD), 191
    XSLT, 194
    XSLT, пример заполнения шаблона, 194
    библиотека LXML (Python), 190
    модель данных с использованием XSD, 192
    основы, 190
    поиск данных с использованием XQuery, 197
  YAML, 184
    документ, пример, 185
    использование в коде Python, 187
    как расширение формата JSON, 189
    комментарий, символ #, 187
    многоточие (...), конец документа, 185
    модель данных, 188
    основы, 184
    пара ключ-значение, 186
    синтаксический анализатор (парсер) формата, 185
    три дефиса (---), начало документа, 185
  обзор, 180
  тип данных, 182
    кортеж, 183
    логическое значение (boolean), 183
    массив (array), 183

Tlgm: @it_boooks

множество, 183
пара ключ-значение, 183
словарь, 183
сложная составная структура, 183
список (list), 183
строка (string), 183
хеш-отображение, 183
хеш-таблица, 183
целое число (integer), 183

Ш

Шаблон, использование для веб-разработки, 210
Шаттлворт, Марк (Mark Shuttleworth), 65
Шлюз (коммутация), 554

Я

Язык моделирования данных, 202

Tlgm: @it_boooks

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Джейсон Эделман, Скотт С. Лоу, Мэтт Осуолт

Автоматизация программируемых сетей

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Снастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 50,05. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**